Optimization for Manhattan's traffic: ridesharing for vehicles for hire

Technische Universiteit Delft
Cognitive Robotics (CoR)

Universitat Politècnica de Catalunya
Centre de Formació Interdisciplinària Superior (CFIS)

Isabel Medrano Sáinz

Bachelor Thesis
Degree in Mathematics - Decree in Engineering Physics

Supervisor: Javier Alonso-Mora (TU Delft)

May, 2020

Abstract

Ridesharing refers to the pooling of several passengers into a vehicle. Its popularity has raised in recent years, as a response to the increase of number of vehicles, rise in traffic congestion and worsening of air quality that have been observed in big and highly populated cities and boroughs such as Manhattan; and as a way to reduce costs for both companies of vehicles for hire and their clients. This work takes as basis *"On-demand high-capacity ride-sharing via dynamic trip-vehicle assignment"* [1], and inserts a pooling parameter to analyze the effect of prioritizing the decrease in total traffic over the quality of service, understood as a function of the delay on the drop off of passengers. Then, it adds a second step, rebalancing, in which vehicles move and idle in order to attend future requests; for different balances between a rise in traffic and a reduction of the number of vehicles used, defined by a rebalancing parameter. Finally, it considers the effects of applying different priorities to these two steps (i.e., the effects of setting different values to these to parameters) on the global situation of Manhattan's roads and the relevant factors such as traffic density, number of vehicles needed to serve requests, vehicle usage or delay and waiting time of passengers.

Keywords optimization, traffic, ridesharing, pooling, rebalancing

AMS2020 90B20

Contents

List of Figures

List of Tables

List of Algorithms

1 Introduction

1.1 Rationale

On an average weekday, Manhattan's vehicles for hire serve almost 500 000 travel requests. There were 250 000 trips performed by taxis in Manhattan daily in 2017, plus 200 000 trips by app-based ride services like Uber, Lyft or Via. This entailed a 15 percent increase in trips attended from the 2013 data. However, this growth was not equally distributed: in fact, taxis lost one third of the trips in that 4 year interval.

The same analysis [2] found that, in that time period: the total number of vehicles for hire increased by 59%, the distance travelled by those vehicles increased by 36%, their average speed decreased by 18% during the day, the total time spent with passengers on board increased by 48%, to a total of 108 000 hours; and the total time spent with no passengers on board increased by 81%, to a total of 66 000 hours.

In summary, while more requests were attended, the growth in number of vehicles and the characteristics of the trips made the whole service less efficient, in terms of actual usage of the existing vehicles. In fact, concern has been raised to whether the situation is sustainable, since the traffic congestion has increased in addition to the worsening of the air quality [3].

Pooling provides a solution to these problems [4], [5]. Uber and Lyft have implement pooling options for their services, so that several unrelated passengers can be attended at once. UberPool has refined its algorithm to the point where the pickup stations have been focused on corners located after traffic lights and situated on the sidewalk to the right of the traffic motion. Moreover, it tries to decrease the amount of corners the vehicle turns, because it increases the travel time [6].

The goal of this work is to estimate the number of vehicles needed to serve these requests, and to study how pooling passengers affects the quality of the service and the state of the traffic.

1.2 State of the art

Analysis for mobility on demand has been made for one passenger per trip [7], and for pooling for several cities [8], [9], [10]. There are multiple approaches to the ridepooling. For example, depending on whether the passengers must have the same initial or ending points, or if stations may be different but in the way of the original route, or if small detours to pick up or drop off other passengers are allowed.

The pooling may be only urban on-demand [11], but it may also occur as a substitution of a commuting [12] or for long-distance travel between cities [13]. The method to compute the assignments varies too: origin and destinations can be paired first by location and time, and then a plan is considered [14]; or collecting points are defined and vehicles periodically visit them [15]; or routes are planned first for the trips and then it checks for other trips along said path [16]. The combination of this options offers different services for diverse targets [17].

1.3 Approach

The approach considered has two steps. The first one is based on *"On-demand high-capacity ride-sharing via dynamic trip-vehicle assignment"* [1]. Its goal is to group passengers into trips, computing this assignment dynamically while the requests are made. The pooling of passengers is promoted, under certain restrictions for the maximum waiting time (time spent by the passenger waiting for the vehicle to pick them up) and delay (time difference between an optimal arrival and the real drop off time). Thus, the cost function is adapted to take into account two opposing goals: decreasing the total delay perceived by the passengers and decreasing the total traffic. This will constitute the first section, *Ridesharing*.

The second step is performed after the whole day has been computed and all passengers have been grouped into trips. It consists on assigning trips to vehicles, and thus, the goal is to establish a plan or schedule for each vehicle, which will determine which trips it will carry out and how it will move in between them. As before, two opposing goals will be taken into account in the cost function: minimizing the number of cars required (which is equivalent to maximize the number of trips to be carried out per vehicle) while trying to avoid the increase in traffic associated to the empty vehicles moving between trips without attending to requests. This will explored in the second section, *Rebalancing*.

2 Ridesharing

2.1 Problem Statement

2.1.1 Definitions

Let (S, d) be a metric space. An individual wanting to travel from $x_r^0 \in S$ to $x_r^1 \in S$ would put in a request r defined by those two points and the time the request was made t_r. This request can be accepted and carried out, or rejected and ignored. Let $\mathscr{R}^*$ be the set of travel requests and let $\mathscr{R}^R$ be the set of rejected requests. The set of travel requests to be attended is: $\mathscr{R} = \{r | r = (x_r^0, x_r^1, t_r)\} = \mathscr{R}^* \setminus \mathscr{R}^R$.

If a given request r is to be attended, then: (i) it will become a passenger once it has been picked up and has not been dropped off, (ii) its optimal arrival time t_r^* would take place if it was picked up at time t_r, and travelled directly between x_r^0 and x_r^1: $t_r^* = t_r + d(x_r^0, x_r^1)$ and (iii) both the pick up t_r^0 and drop off time t_r^1 must hold maximum waiting time Δ_w and maximum delay Δ_d constraints:

$$t_r \leq t_r^0 \leq t_r + \Delta_w \tag{1}$$

$$t_r^* \leq t_r^1 \leq t_r^* + \Delta_d \tag{2}$$

Requests are carried out by vehicles. A vehicle v is defined by its capacity ν, i.e., the total number of passengers it can carry at once, its position x_v, the set of passengers on board P_v, the set of requests it plans to carry out R_v, and its scheduled plan Γ_v. The scheduled plan contains the path the vehicle will follow, and it is defined by the order in which the pick ups and drop offs must be performed. This plan must have $|P_v| + 2 \cdot |R_v|$ stops, since all passengers must be dropped off once and all requests must be picked up and then dropped off. For a vehicle plan to be feasible, it must fulfill:

(c1) the maximum waiting time and delay constraints (equations 1 and 2), $\forall r \in R_v$,

(c2) the maximum delay constraint (equation 2), $\forall r \in P_v$,

(c3) $\forall r \in R_v$, r is picked up before being dropped off,

(c4) at any given moment of the plan, the number of passengers at that point must be smaller than the capacity ν. In particular, $|P_v| \leq \nu$. Let Γ_v be the ordered plan, that contains $\{+i, -i \mid \forall i \in R_v\}, \{-i \mid \forall i \in P_v\}$ pick ups and drop offs of requests and drop offs of passengers respectively. Let $\Gamma_v(j)$ be the j-th element of the plan. Then:

$$\sigma(j) = |P_v| + \sum_{k=1}^{j} \text{sign}(\Gamma_v(k)) \leq \nu, \ \forall j, 1 \leq j \leq |P_v| + 2 \cdot |R_v| \tag{3}$$

Obs: this higher bound holds for all subsums, and not just the ones that start from the beginning: $\sigma(j_2) - \sigma(j_1) = \sum_{k=j_1}^{j_2} \text{sign}(\Gamma_v(k)), \ \forall j_1, j_2, 1 \leq j_1 < j_2 \leq |P_v| + 2 \cdot |R_v|$

Obs: (c3) guarantees that $\sigma(j) \geq 0, \forall j, 1 \leq j \leq |P_v| + 2 \cdot |R_v|$. The reciprocal is not true.

The fleet of vehicles $\mathscr{V} = \{v | v = (x_v, P_v, R_v, \Gamma_v)\}$ defines a global plan, $\Gamma_{\mathscr{V}}$, which must also fulfill some conditions:

(c5) all plans within it, $\{P_v | v \in \mathscr{V}\}$, are feasible,

(c6) all requests $\mathscr{R}^*$ are either accepted or rejected: $\mathscr{R}^* = \mathscr{R} \amalg \mathscr{R}^R$

(c7) all accepted requests must be carried out by a vehicle:

$$\bigcup_{v \in \mathscr{V}} R_v = \mathscr{R} \tag{4}$$

(c8) a request must be carried out by, at most, one vehicle:

$$R_{v_1} \cap R_{v_2} = \emptyset, \forall v_1, v_2 \in \mathscr{V}, v_1 \neq v_2 \tag{5}$$

(c9) a passenger must be carried out by one and only one vehicle:

$$P_{v_1} \cap P_{v_2} = \emptyset, \forall v_1, v_2 \in \mathscr{V}, v_1 \neq v_2 \tag{6}$$

2.1.2 Cost function

There are three main goals when defining a global plan:

1. Avoid rejecting requests
2. Minimize the total delay suffered by passengers
3. Minimize the traffic

However, these goals have conflicting effects. For example, by rejecting requests and pooling passengers, the traffic gets reduced; while by assigning only one request per vehicle as soon as the request is made, the delay gets reduced to zero, but the fleet size required and the associated traffic increase.

Let c_r be the cost associated to rejecting a request, and let $C(v)$ be the cost of plan Γ_v, which defines pick up and drop off times for all its passengers and requests. Then, the cost of the global plan is:

$$\mathscr{C}(\Gamma_{\mathscr{V}}) = \sum_{r \in \mathscr{R}^R} c_r + \sum_{v \in \mathscr{V}} C(v) \tag{7}$$

For a given passenger, let $\delta_d(r)$ be its total delay given by t_r^1 defined by its vehicle plan:

$$\delta_d(r) = t_r^1 - (t_r^* + d(x_r^0, x_r^1)) \tag{8}$$

Let $s(i), i \in \Gamma_v$ be the station at which order $\Gamma_v(i)$ takes place, whether it is a pick up or a drop off. Assuming that trips end at the station where the last drop off takes place; that the plans get updated so that only future stops are considered; and that, if the vehicle has just been defined, then it starts at the station where the first pick up takes places, so $x_v = s(\Gamma_v(1)))$. Then, the total travelling time is:

$$\delta_t(v) = d\big(x_v, s(\Gamma_v(1))\big) + \sum_{i=1}^{|\Gamma_v|-1} d\big(s(\Gamma_v(i)), s(\Gamma_v(i+1))\big) \tag{9}$$

Now, the cost of each of the plans needs to take into account the trade-off between the increase (or decrease) of the delay, with the decrease (or increase) of the traffic. Let $\alpha \in (0, 1)$ be the pooling parameter, so that if $\alpha = 1$, the main concern is decreasing the traffic while ignoring the delay it may incur on the passengers (within the constrictions of equations 1 and 2). The cost of plan Γ_v is:

$$C(v) = (1 - \alpha) \cdot \sum_{r \in P_v \cup R_v} \delta_d(r) + \alpha \cdot \delta_t(v) \tag{10}$$

The global cost function is:

$$\mathscr{C}(\Gamma_{\mathscr{V}}) = c_r \cdot |\mathscr{R}^R| + (1 - \alpha) \cdot \sum_{r \in \bigcup_{v \in V} P_v \cup R_v} \delta_d(r) + \alpha \cdot \sum_{v \in \mathscr{V}} \delta_t(\Gamma_v) \tag{11}$$

2.1.3 Dynamic problem

If the data for all requests to be made during the day was available at the beginning of that day, then the cost of the global plan could be minimized, subjected to the constraints stated on the previous sections and the size of the available fleet. However, its size makes this problem unmanageable. Moreover, the data of the requests made is only available afterwards, and therefore, a dynamic implementation must be applied. Once that a first plan has been draft assuming that all requests are new requests, the goal is to update that global plan as new requests appear by minimizing its cost.

Requests are assumed to take place in a map built as a weighted graph and are to be received in intervals of Δ_{req} seconds. The information is saved, so at time $t = n\Delta_{req}$, all requests made before t are known.

New requests can be rejected, attended by existing vehicles, or attended by new vehicles. It is assumed that new vehicles can be created at each station at any given moment of time, but a maximum number of vehicles to be considered for each request is established.

Requests accepted in previous iterations must be carried out, however, the plan for them - vehicle they are assigned to, pick up time, etc. - may change.

2.2 Algorithm

2.2.1 Trips

Let a trip be a set of requests that can be carried out by the same vehicle (for a certain initial vehicle position) while fulfilling the capacity, maximum waiting and delay constraints.

It is immediately deduced that any subset of a trip is also a trip, and that, indeed, all subsets of a trip must be trips in order for the set to be a trip. The reciprocal is not hold: even if all subsets of a set of requests are trips, this does not imply that said set is a trip. Nevertheless, this property can be used to build all possible trips given the existing requests.

Let T_i be the set of trips of size i. First, by considering each individual request as a trip, the set of trips of size 1 can be build: $T_1 = \{ \{r\} \mid r \in \mathscr{R}\}$. The set of trips of size 2 is then build by checking if at least one of the six possible plans for two requests $(+1,-1,+2,-2)$, $(+1,+2,-1,-2)$, $(+1,+2,-2,-1)$, $(+2,-2,+1,-1)$, $(+2,+1,-2,-1)$, $(+2,+1,-1,-2)$ is feasible.

Once T_k has been computed, consider all pairs of its elements $t_1, t_2 \in T_k$ that only differ on one element: $|t_1 \cup t_2| = k+1$ ($|t_1 \cap t_2| = k-1$). If $t_1 \cup t_2 = \{r_1, \cdots, r_{k+1}\}$, then only if $(t_1 \cup t_2) \setminus r_i \in T_k, \forall i$, the trip $(t_1 \cup t_2)$ could be feasible. Therefore, the amount of trips whose feasibility needs to be checked is reduced drastically: otherwise, plans for all combinations of k elements would need to be checked.

2.2.2 Plans

A plan is the result of associating a trip with a vehicle, and defining the order of the pick ups and drop offs. Since a vehicle may already have passengers, the plan also needs to take them into account.

In the actual implementation, computing all possible plans may take too long. The magnitude of the number of possible plans is:

- Let n be the total number of requests to be attended. The actions or orders of the plan consist on either to pick up or drop off a passenger. The number of ways these actions can be ordered (with no restrictions) is $(2n)!$.

  However, a plan is only feasible if pick ups take place before drop offs (overlooking for now the capacity restriction). These permutations can be grouped into classes comprised of elements which are only different in that a request's pick up is swapped with its own drop off (example: $[(+1,+2,-1,-2)] = \{(+1,+2,-1,-2),(+1,-2,-1,+2),(-1,+2,+1,-2),(-1,-2,+1,+2)\}$). One and only one element of each class is feasible.

  The total number of elements in a class is 2^n (choosing if $+$ or $-$ comes first for each request, so choosing n times between 2 options). Therefore, the total number of plans where, for all requests, the pick up comes before the drop off is:

$$\left| \left\{ \text{ Plans with } (t_r^0 < t_r^1), \forall r \right\} \right| = \frac{(2n)!}{2^n} \tag{12}$$

  The same result can be achieved by considering that a plan has $2n$ spots to be filled and that, consecutively, the two positions for the pick up and drop off pair for each request can be chosen out of the spots that remain empty.

  There are $\binom{2n}{2}$ ways of placing the first pair of orders; then, there are $\binom{2n-2}{2}$ ways of placing the second pair of orders; etc, until only a pair of orders remain, to be located on the last two empty spots $\binom{2}{2}$.

  By replacing each binomial coefficient with its factorial expression, the total number of plans where, for all requests, the pick up comes before the drop off is:

$$\left| \left\{ \text{ Plans with } (t_r^0 < t_r^1), \forall r \right\} \right| = \prod_{i=0}^{2n-2} \binom{2n-i}{2} = \frac{(2n)!}{2!(2n-2)!} \frac{(2n-2)!}{2!(2n-4)!} \cdots \frac{2!}{2!0!} = \frac{(2n)!}{2^n} \tag{13}$$

  which yields the same result as the one obtained on equation 12.

- How many ways of organizing a plan are there, so that there are always between 0 and ν passengers?

  If there are no capacity limitations, the problem is analogous to the monotonous path: starting with an empty vehicle, if pick ups are understood as steps up and drop offs as steps down, then the number of passengers at any moment in time is the height of the mountain range at that point. The number of passengers must be non-negative, therefore, the path must always be above the zero line. The number of paths of length n, C_n is given by the Catalan numbers.

  Let l be the number of passengers carried before the car empties for the first time, i.e., let $2l$ be the first point of the mountain range at which the height goes back to zero. The first step is always up, and the $2l$-th step must be down, so the path between steps 2 and $2l-1$ is actually a monotonous path with its baseline at height one. Therefore, there are C_{l-1} ways to reach $2l$ without having touched the baseline at height zero. The rest of the path until reaching $2n$ is a mountain range of length $2n-2l$, so there are C_{n-l} ways of completing it. Taking into account that C_0=1 and l can range between 1 and n, then:

$$C_n = \sum_{l=1}^{n} C_{l-1} C_{n-l} \tag{14}$$

$\boldsymbol{n}$	0	1	2	3	4	5	6	7	8	9	10
$\boldsymbol{C_n}$	1	1	2	5	14	42	132	429	1430	4862	16796

Table 1: C_n for $n \leq 10$

A similar computation can be made for mountains range with a height limit, i.e., trips with a capacity limit ν. Let $C(n,\nu)$ be the number of different mountain ranges of length $2n$ (n steps up and n steps down) with height limit ν:

- $\nu \geq n \Rightarrow$ All passengers would fit at once, so it is equivalent to not applying any capacity restrictions. Therefore, $C(n,\nu) = C_n, \forall n \leq \nu$.
- $\nu = n-1 \Rightarrow C(n,n-1) = C_n -$ { mountains ranges of height n }. There is only one path of length $2n$ and height n: going n steps up and then going n steps down. Therefore, $C(n,n-1) = C_n - 1, \forall n$.
- $\nu = 1 \Rightarrow$ each passenger is picked up and dropped off individually, so the heights list is $(0,1,0,\cdots,0,1,0)$. Therefore, $C(n,1) = 1, \forall n$.
- $\nu = 2 \Rightarrow$ if at a certain point of the mountain range, the height is 0, then necessarily, the next step must be up to a height of 1; and if at a certain point of the mountain range, the height is 2, then necessarily, the next step must be up to a height of 1. Since from height 1, in one step, only heights 0 or 2 are reachable, then, the the heights list is: $h(0) = h(2n) = 0; h(2k-1) = 1, \forall 1 \leq k \leq n; h(2k) = \{0,2\}, \forall 1 \leq k \leq n-1$. Thus, there are $n-1$ points at which the mountain range "chooses" between two possibilities. Therefore, $C(n,2) = 2^{n-1}, \forall n$.

  $C(n,2) = 2^{n-1} = \sum_{i=0}^{n-1} \binom{n-1}{i}$, where i is the number of times height 2 is reached.
- For all other cases, a recursive expression similar to the one found for the Catalan numbers can be determined. Let l be the number of passengers carried before the vehicle with capacity ν empties for the first time, i.e., let $2l$ be the first point of the mountain range with height limit ν, at which the height goes back to zero. The first step is always up, and the $2l$-th step must be down, so the path between steps 2 and $2l-1$ is actually a monotonous path with its baseline at height one, and relative maximum capacity $\nu-1$. Therefore, there are $C(l-1,\nu-1)$ ways to reach $2l$ without having touched the baseline at height zero and keeping the capacity restriction. The rest of the path until reaching $2n$ is a mountain range of length $2n-2l$ with capacity ν, so there are $C(n-l,\nu)$ ways of completing it.

  Taking into account that $C(1,\nu) = C(0,\nu) = 1, \forall \nu$, $C(n,1) = 1, \forall n$, and l can range between 1 and n; then, $\forall n$:

$$C_n = \sum_{l=1}^{n} C(l-1,\nu-1) C(n-1,\nu) \tag{15}$$

The results for the first few pairs are represented on table 2. This computation does not take into account the possible orders: it assumes that the pick ups are performed in one of the $n!$ possible orders (which can be fixed by multiplying by a factor of $n!$, as seen in table 3) and disregards the order in which drop offs can actually take place.

		n					
		0	**1**	**2**	**3**	**4**	**5**
ν	**1**	1	1	1	1	1	1
	2	1	1	2	4	8	16
	3	1	1	2	5	13	34
	4	1	1	2	5	14	41
	5	1	1	2	5	14	42

Table 2: $C(n, \nu)$ for $n \leq 5$ and $\nu \leq 5$

		n					
		0	**1**	**2**	**3**	**4**	**5**
ν	**1**	1	1	2	6	24	120
	2	1	1	4	24	192	1800
	3	1	1	4	30	312	4080
	4	1	1	4	30	336	4920
	5	1	1	4	30	336	5040

Table 3: $C(n, \nu)n!$ for $n \leq 5$ and $\nu \leq 5$

Figure 1 shows the different mountain ranges for $n = 4$, grouped by its maximum vehicle usage. The set of mountain ranges of a given capacity ν contains the set of mountain ranges whose maximum vehicle usage is smaller than ν.

Figure 2 shows the difference between how fast these different quantities - permutations, C_n, $C(n, v)$, etc. - grow with the size of the number of passengers being considered, n.

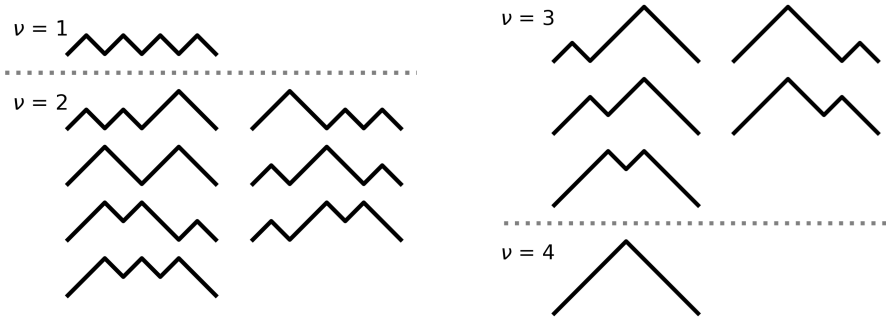

Figure 1: Mountain ranges with $n = 4$ by capacity

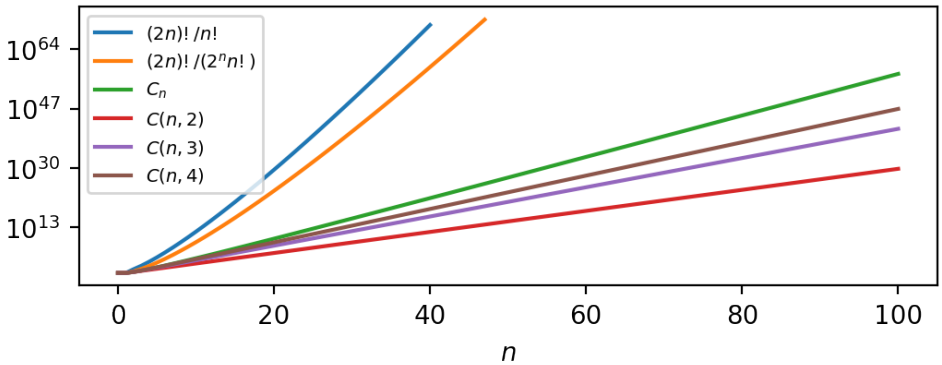

Figure 2: Difference between growth rates of $\frac{(2n)!}{n!}, \frac{(2n)!}{2^n \cdot n!}, C_n, C(n, \nu)$

- From the previous computation, all possible feasible plans $\widetilde{C}(n, \nu)$ (that take into account both conditions) can be computed. Assuming the pick up order is fixed:

  If $\nu \geq n$, then the number of feasible plans is not restricted by the capacity, and the result is the same as the one found in the first item: $\widetilde{C}(n, \nu) = \frac{(2n)!}{2^n \cdot n!}, \forall n$. If $\nu = n - 1$, then the $n!$ possible drop off orders (for the mountain rage that reaches height n) must be subtracted from $\widetilde{C}(n, n)$: $\widetilde{C}(n, n-1) = \frac{(2n)!}{2^n \cdot n!} - n!, \forall n$.

For $\nu = 1$, $\widetilde{C}(n,1) = 1, \forall n$. For $\nu = 2$, the number of feasible paths given a mountain range depends on the number of times height 2 is reached, since that is the only time at which there are several possible orders (otherwise, the pick up order is already scheduled, the pick up/drop off structure is already given by the mountain range shape, and if the height is 1, there is only one passenger to drop off). There are $\binom{n-1}{i}$ paths that reach i times height 2 i, and at each of those points, one out of two passengers needs to be chosen to drop off, therefore: $\widetilde{C}(n,2) = \sum_{i=0}^{n-1} \binom{n-1}{i} 2^i, \forall n$.

Otherwise, consider a mountain range that represents a vehicle with n passengers and maximum capacity ν, defined by the number of consecutive steps up or down taken: $(u_1, d_1, u_2, d_2, \cdots, u_s, d_s)$. Let $\sigma_\downarrow(i) = \sum_{k=1}^{i}(u_k + d_k)$ and let $\sigma_\uparrow(i) = \sigma_\downarrow(i-1) + u_i$, then: $0 \leq \sigma_\downarrow(i) < \sigma_\uparrow(i) \leq \nu$.

Assuming that the order of the pick ups is fixed, for a certain mountain range (structure of pick ups and drop offs) the total number of feasible plans is:

$$\prod_{\text{steps}} \binom{\text{current height}}{\#\text{steps down}} (\#\text{steps down})! = \prod_{i=1}^{s} \binom{\sigma_\uparrow(i)}{d_i} d_i! = \prod_{i=1}^{s} \frac{(\sigma_\uparrow(i))!}{(\sigma_\downarrow(i))!} \tag{16}$$

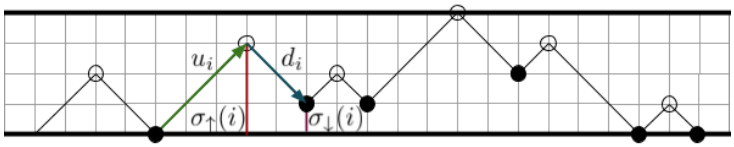

Figure 3: #Plans with a given shape

An example for a mountain range with $n = 11$ and $\nu = 4$ can be found on figure 3. The plan shown in the figure is $(2, 2, 3, 2, 1, 1, 3, 2, 1, 3, 1, 1)$. By applying the previous formula, the number of plans: $\frac{2!}{0!}\frac{3!}{1!}\frac{2!}{1!}\frac{4!}{2!}\frac{3!}{0!}\frac{1!}{1!} = 1728$

By adding these numbers for all possible mountain ranges, the total number of feasible plans arises, as shown in table 4. This computation assumes that the pick ups are performed in one of the $n!$ possible orders, which can be fixed by multiplying by a factor of $n!$, as seen in table 5.

		n					
		0	**1**	**2**	**3**	**4**	**5**
ν	**1**	1	1	1	1	1	1
	2	1	1	3	9	39	81
	3	1	1	3	15	81	387
	4	1	1	3	15	105	825
	5	1	1	3	15	105	945

Table 4: $\widetilde{C}(n,\nu)$ for $n \leq 5$ and $\nu \leq 5$

		n					
		0	**1**	**2**	**3**	**4**	**5**
ν	**1**	1	1	2	6	24	120
	2	1	1	6	54	936	9720
	3	1	1	6	90	1944	46440
	4	1	1	6	90	2520	319275
	5	1	1	6	90	2520	365715

Table 5: $\widetilde{C}(n,\nu)n!$ for $n \leq 5$ and $\nu \leq 5$

All those series grow much slower than the number of permutations, but it is still unfeasible to compute them all, for all trips, for all vehicles (since depending on its initial position, the plan with the lowest cost may vary), for all iterations.

Trips, as seen before, are computed in increasing size. Once a plan has been established for a vehicle-trip pair and a new request wants to be added:

- if the total number of passengers and requests of the plan does not reach the capacity, then all possible plans with the added request are computed, and the one with smallest cost is chosen;
- else, if the capacity is already reached or exceeded, then the internal order of the plan is kept and the pick up and drop off of the new request are planned between previous actions. For example, for $\nu = 2$ with current plan $(+1, +2, -1, -2)$, the 3rd passenger's pick up may be established in one of five positions: before $+1$, after -2 and between actions.

This greatly reduces the computational time.

2.2.3 ILP problem

The problem can also be interpreted as an ILP. Let $\epsilon_{ij}, i \in T, j \in R$ be a binary variable that takes value 1 if trip i is assigned to vehicle R and 0 otherwise, and let $\tilde{\epsilon}_k, k \in R$ be a variable that takes value 1 if k belongs to one of the trips assigned, $k \in \bigcup_{i \in T} i \cdot \epsilon_{ij}$. Let $\varepsilon = \{\epsilon_{ij}, \tilde{\epsilon}_k\}$ and T_r the set of trips that contains request r. Then, the problem results in:

$$\begin{array}{lll} \min_{\varepsilon} & \mathscr{C}(\varepsilon) & \\ s.t.: & \sum_{i \in T} \epsilon_{ij} \leq 1 & \forall j \in V \\ & \sum_{i \in T_r, j \in V} \epsilon_{ij} + \tilde{\epsilon}_k = 1 & \forall k \in R \end{array} \tag{17}$$

where the cost function is:

$$\mathscr{C}(\varepsilon) = \sum_{(i,j) \in TV} \epsilon_{ij} \cdot \left((1-\alpha) \cdot \sum_{r \in i} \delta_d(r) + \alpha \cdot \delta_t(j) \right) + \sum_{k \in R} \tilde{\epsilon}_k \cdot c_{rej} \tag{18}$$

ILP Implementation

In order to solve the integer linear problem, the Mosek solver is used [18].

Optimality

The result obtain may not be optimal: the computation of feasible trips or a feasible solution for the ILP may be stopped before that solution has been found if it is taking too long; moreover, not all plans are considered for trips with more total number of passengers and requests than the capacity, as explained in previous sections.

3 Rebalancing

3.1 Problem statement

Let $\mathscr{T}$ be the set of trips obtained as a solution for the ridesharing problem stated in the previous section *ILP problem*; said trips, compromised by a set of requests, can be understood as individual travels between stations and can be described as follows:

$$\begin{aligned} \mathscr{T} = \{\pi | \pi = \{i, s_0, s_1, t_0, t_1\}\} \quad , \\ \pi.i \quad & \pi\text{'s vehicle identifier} \\ \pi.s_0 \quad & \pi\text{'s initial station} \\ \pi.s_1 \quad & \pi\text{'s final station} \\ \pi.t_0 \quad & \pi\text{'s time of first pick-up} \\ \pi.t_1 \quad & \pi\text{'s time of last drop-off} \end{aligned} \tag{19}$$

The ridesharing problem in equation 17 is stated under the assumption that vehicles can appear and disappear from each station at any given moment. This assumption is made because it is presumed that there are enough vehicles to have one at any given station in a very short time interval, and so that only vehicles carrying passengers need to be considered. The rebalancing problem's goal is to assign the same vehicle to several different trips, without adding delay or waiting time to the trip's passengers (feasibility condition), and therefore building long-term plans for vehicles and reducing the total fleet required. However, this leads to an increase to the global traffic and the total distance driven.

In order to incorporate this trade-off in the cost function, two new variables are defined: $\beta \in [0, 1]$ and $\delta \geq 0$ set the "amount" of rebalancing taking place (higher β, δ implies a higher harnessing of vehicles).

3.1.1 Interpretation as an ILP

$$\begin{aligned} & \text{argmin}_{x_{ij}} \left\{ \sum_{ij} x_{ij} \left((1-\beta) \cdot c_{ij} - \beta \cdot \delta \right) \right\} \\ & \text{subjected to:} \quad \text{(i)} \quad x_{ij} \in \{0, 1\}, \forall i, j \\ & \qquad\qquad\quad \text{(ii)} \quad \sum_{1 \leq j \leq t} x_{ij} \leq 1, \forall i \\ & \qquad\qquad\quad \text{(iii)} \quad \sum_{1 \leq j \leq t} x_{ji} \leq 1, \forall i \end{aligned} \tag{20}$$

where c_{ij} is the time it takes to travel from $i.s_1$ to $j.s_0$ and x_{ij} is a binary variable that takes value $x_{ij} = 1$ if trip i's vehicle is to perform trip j immediately after dropping-off trip i's last passenger, and value 0 otherwise. Condition (i) ensures this variable to be binary, while conditions (ii) and (iii) guarantee that vehicles are assigned to at most one trip.

3.1.2 Interpretation as an Assignment Problem

The assignment problem is an optimization problem that, given a weighed bipartite graph, computes a matching between the two sides of the graph that minimizes the global cost.

Let $\mathscr{T}$ be the set of trips, from which the set of starting and ending points can be defined:

$$\begin{aligned} T_0 = \{(t_0, s_0) | \exists s \in \mathscr{T} \text{ with } s.t_0 = t_0, s.s_0 = s_0\} \\ T_1 = \{(t_1, s_1) | \exists s \in \mathscr{T} \text{ with } s.t_1 = t_1, s.s_1 = s_1\} \end{aligned} \tag{21}$$

The assignment problem is said to be balanced if both sides of the graph have the same number of nodes. These three sets have the same cardinal $|\mathscr{T}| = |T_0| = |T_1|$, and therefore, the global case is balanced. However, if the problem is restricted to a time interval, the amount of trips that end in said interval may be different to the number of trips that start in it. In this case, the assignment problem is unbalanced.

Let $G = (V, E)$ be a graph with vertices $V = T_0 \cup T_1$ and edges $E = \{e_{ij} = (i, j) | i \in T_1,\ j \in T_0,\ j.t_0 \geq i.t_1 + c_{ij}\}$ (bipartite undirected graph), where c_{ij} is the travelling time between $i.s_1$ and $j.s_0$ and with associated costs $\forall e_{ij} \in E$, $C_{ij} = (1 - \beta) \cdot c_{ij} - \beta \cdot \delta$ (weighted graph).

3.1.3 Interpretation of δ and other properties

Optimal value *The minimum of the cost function is a non-positive value.*

Proof. If $x_{ij} = 0, \forall i, j$, then $\sum_{ij} x_{ij} \left((1-\beta)\cdot c_{ij} - \delta \cdot \beta\right) = 0$. Therefore
$\min_{x_{ij}} \left\{\sum_{ij} x_{ij} \left((1-\beta)\cdot c_{ij} - \beta \cdot \delta\right)\right\} \leq 0.$ □

Interpretation of δ *The value of δ determines the maximum travelling time allowed in the rebalancing. Moreover, for $\beta \neq 1$ this travelling time is given by:*

$$c \leq \frac{\beta \cdot \delta}{1-\beta}$$

Proof. First, let's proof that, given a possible assignment $i \to j$ in an optimal solution, if $((1-\beta)\cdot c_{ij} - \beta \cdot \delta) > 0$, then $x_{ij} = 0$.
Let's assume otherwise, i.e., that exists an optimal assignment $X^* = \{x_{ab} | a, b \in \mathscr{T}\}$ such that $\exists x_{ij} \in X^* : x_{ij} = 1$ and $((1-\beta)\cdot c_{ij} - \beta \cdot \delta) > 0$. Let $\tilde{X}^* = \{\tilde{x}_{ab} | a, b \in \mathscr{T}, \tilde{x}_{ab} = x_{ab} \in X^*, \forall (a,b) \neq (i,j), \tilde{x}_{ij} = 0\}$.
Let $\mathscr{S} = \sum_{\{x_{ab} \in X^*\} \setminus x_{ij}} x_{ab} \left((1-\beta)\cdot c_{ab} - \beta \cdot \delta\right)$:

$$\left.\begin{array}{l} S^* = \sum_{x_{ab} \in X^*} x_{ab} \left((1-\beta)\cdot c_{ab} - \delta \cdot \beta\right) = x_{ij} \left((1-\beta)\cdot c_{ij} - \beta \cdot \delta\right) + \mathscr{S} = \left((1-\beta)\cdot c_{ij} - \beta \cdot \delta\right) + \mathscr{S} \\ \tilde{S}^* = \sum_{x_{ab} \in \tilde{X}^*} x_{ab} \left((1-\beta)\cdot c_{ab} - \delta \cdot \beta\right) = x_{ij} \left((1-\beta)\cdot c_{ij} - \beta \cdot \delta\right) + \mathscr{S} = \mathscr{S} \end{array}\right\} \Rightarrow$$

$\Rightarrow S^* > \tilde{S}^* \Rightarrow X^*$ is not an optimal assignment for the minimization problem, and thus the assumption is wrong. Therefore:

$$x_{ij} = 1 \implies (1-\beta)\cdot c_{ij} - \beta \cdot \delta \leq 0 \iff (1-\beta)\cdot c_{ij} \leq \beta \cdot \delta$$

$$\text{If } \beta \neq 1 \implies c_{ij} \leq \frac{\beta \cdot \delta}{1-\beta}$$

The limit for $\beta \to 1$, yields that no travelling time limit applies:

$$\lim_{\beta \to 1^-} \frac{\beta \cdot \delta}{1-\beta} = \infty$$

□

Case $\beta = 0$ *The only possible assignments are those with no travel involved (no increase in the total distance driven), and therefore, it is enough to solve the problem individually for each station.*

Proof. $\beta = 0 \implies \text{argmin}_{x_{ij}} \left\{\sum_{ij} x_{ij} \left((1-\beta)\cdot c_{ij} - \beta \cdot \delta\right)\right\} = \text{argmin}_{x_{ij}} \left\{\sum_{ij} x_{ij} \cdot c_{ij}\right\}$
Since both x_{ij} and c_{ij} take non-negative values: $\min_{x_{ij}} \left\{\sum_{ij} x_{ij} \cdot c_{ij}\right\} \geq 0$.
But from the previous proposition, the minimum value is non-positive. Therefore:

$$\min_{x_{ij}} \left\{\sum_{ij} x_{ij} \cdot c_{ij}\right\} = 0 \implies \forall ij, \text{either } x_{ij} = 0 \text{ or } c_{ij} = 0$$

$x_{ij} = 0, \forall i, j$ is the solution from the ridesharing problem, so the solution being sought is the one with the maximum number of $x_{ij} \neq 0$.
$x_{ij} \neq 0 \implies c_{ij} = 0 \iff i.s_1 = j.s_0$ and therefore the only assignments allowed are those within the same station, and thus, the problem to solve is, given the set of stations $\mathscr{S}$:

$$\text{argmax}_{x_{ij}} \left\{\sum_{ij} x_{i,j} \middle| i.s_1 = j.s_0\right\} = \sum_{s \in \mathscr{S}} \text{argmax}_{x_{ij}} \left\{\sum_{ij} x_{i,j} \middle| i.s_1 = j.s_0 = s\right\}$$

□

3.2 Hungarian Algorithm

3.2.1 Algorithm

In order to solve the assignment problem stated in equation 20, a graph implementation of the Hungarian Algorithm [19] is used. Given a weighted bipartite graph $G = (N, E), N = X \cup Y, X \cap Y = \emptyset, |X| = |Y|$ with an associated cost function $c : X \times Y \to \mathbb{R}_{\geq 0}$ that defines the weight or cost of each edge, the goal is to find a matching m, i.e., a bijective function that defines a one-on-one correspondence between X and Y, that minimizes the total cost of the assignment $C = \sum_{x \in X} c\ (x, m(x))$, with $(x, m(x)) \in E$.

A function $l : N \to \mathbb{R}$ is a label if $\forall i \in X, j \in Y, l(x) + l(y) \leq c\ (x, y)$. The equality graph defined by this label function is $G' = (N, E')$ where the only edges defined are those in which the equality holds $E' = \{(x, y) \in X \times Y \mid l(x) + l(y) = c(x, y)\}$.

Optimal assignment *Let l be a label and G' its equality graph. If there exists a matching m between X and Y within G', then it is an optimal assignment.*

Proof. Let m' be a matching between X and Y.

$$\text{cost}(m') = \sum_{i \in X} c\ (i, m'(i)) \geq \sum_{i \in X} l(i) + l(m'(i)) = \sum_{i \in X} l(i) + \sum_{i \in Y} l(i) = \text{cost}(m)$$

□

A path is alternating if it contains edges that alternate between belonging and not belonging to the already defined assignment, and a path is augmenting if it is alternating and both of its ends are yet unassigned.

The Hungarian Algorithm looks for a matching within the equality graph defined by a label that gets updated by looking for augmenting paths that start from unassigned nodes and belong to the said equality graph.

Algorithm 1: Hungarian Algorithm pseudo code

```
Input  : graph G, cost matrix c
Output: matching m

Initialize label l
Initialize equality graph G'
Initialize matching m

while ∃i ∈ X unassigned do
    look for augmenting path
    if found then
        update match
    else
        update labels l_X, l_Y
        update equality graph EG
    end
end
```

If an augmenting path is not found from node $i \in X$, then the label l is updated into l'. E' must be updated too. Let $X' \subset X$ and $Y' \subset Y$ be the set of nodes explored while trying to build the augmenting path, i.e. the set of nodes that belong to an alternating path within G' starting from i.

$$\delta = \max_{x \in X', y \in Y \setminus Y'} \{l(x) + l(y) - c\ (x, y)\} \tag{22}$$

$$l'(i) = \begin{cases} l(i) - \delta & \text{if } i \in X' \\ l(i) + \delta & \text{if } i \in Y' \\ l(i) & \text{otherwise.} \end{cases} \quad , \forall i \in N \tag{23}$$

Algorithm 2: Loop from Hungarian Algorithm

Input : graph G, cost matrix c, current label l, equality graph G' and matching m, $i \in X$
Output: if an augmented path was found; updated label l, equality graph G' and matching m

```
// Look for an augmenting path
```
$k \leftarrow i$, $exp \leftarrow \{i\}$, $X' \leftarrow \{i\}$, $Y' \leftarrow \emptyset$, $dist \leftarrow dist : dist(i) = 0, dist(j) = \infty, \forall j \in Y, \forall j \in X \setminus \{i\}$
while $exp \neq \emptyset$ **do**
  $k \leftarrow exp.next(),\ exp.pop()$
  if $k \in X$ **then**
    $S \leftarrow \{j \in Y | (i,j) \in G', j \notin Y'\}$
    if $S \neq \emptyset$ **then**
      $exp.add(S)$, $Y'.add(S)$
      $dist(j) \leftarrow dist(k) + 1, \forall j \in S$
    end
  else
    if k *assigned* **then**
      $exp.add(m^{-1}(k)),\ X'.add(m^{-1}(k)),\ dist(m^{-1}(k)) \leftarrow dist(k) + 1$
    else
      // Augmenting path found. Update match
      $d \leftarrow dist(k)$, $y_m \leftarrow k$
      while $d \geq 0$ **do**
        $x_m \leftarrow \{x \in X | dist(x) = d - 1, (x, y_m) \in G'\}.any(),\ d = d - 2$
        $m(x_m) = y_m,\ m(y_m) = x_m$
        $y_m \leftarrow \{y \in Y | dist(y) = d, (x_m, y) \in G'\}.any()$
      end
      return True
    end
  end
end

```
// Augmenting path does not exist. Compute delta...
```
$\delta \leftarrow -\infty$
for $x \in X'$ **do**
  for $y \notin Y'$ **do**
    if $\delta < l(x) + l(y) - c(x,y)$ **then**
      $\delta \leftarrow l(x) + l(y) - c(x,y)$
    end
  end
end

```
// ...and update the label/G'
```
for $x \in X'$ **do**
  $l(x) \leftarrow l(x) - \delta$
end
for $y \in Y'$ **do**
  $l(y) \leftarrow l(y) + \delta$
end
$G' \leftarrow \emptyset$
for $x \in X'$ **do**
  for $y \in Y'$ **do**
    if $c(x,y) = l(x) + l(y)$ **then**
      $G'.add((x,y))$
    end
  end
end
return False

Notes

- Notation: *exp* is a FIFO-type structure like a queue, so that *next*() reads its first element, *pop*() deletes said first element and *add*() appends a new element. *A.any*() chooses any element of set A, or *nan* if $A = \emptyset$.
- G' must be re-computed every iteration. Usually, updating the label will add one (or several) edges to E', but there are some exceptions in which some edges will be destroyed, as seen on figure 4.

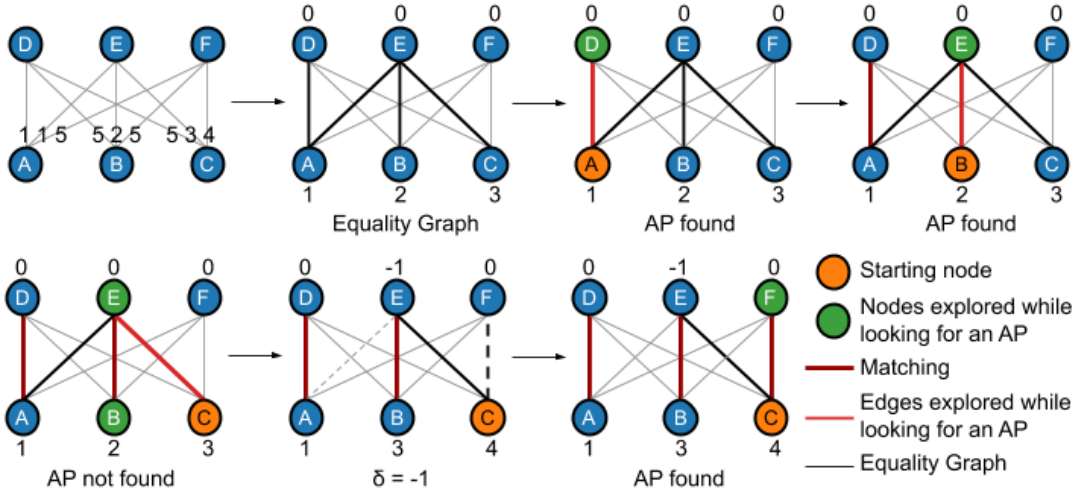

Figure 4: Hungarian Algorithm: need of recompute G' every iteration

- Equality graph exploration. The goal is to find an augmenting path from a yet unmatched element $x_0 \in X$. Since x_0 is not assigned yet, then the first edge of the augmenting path will be an unmatched edge that will lead to an element on the other side of the bipartition. Therefore, all elements from Y connected to x_0 will be visited. Now, for each one of all those nodes, it either is unmatched and an augmenting path of length one has been found, or it is matched, in which case - since the next edge from the augmenting path must belong to the assignment - only the element from X to which it is matched will be visited. By doing this recursively, X' and Y' can be build.

  Therefore, when exploring the equality graph for an augmenting path: if the current node belongs to X, add to Y' all nodes from Y that are connected to the current node, and if it belongs to Y, then if it exists, add its match to X'.

  When testing the algorithm, a breadth-first search worked faster than a depth-first search for the exploration of G'. This may be due to the fact that a BFS does not take into account the order in which the nodes are given; with the DFS, it prioritized the search starting with the first elements, which where the first to be assigned, so it re-computed the same exploration than ones starting with other elements. While this could be fixed by randomizing the choosing of the element to look further into, the BFS worked fine, so no further attempts were made with a DFS.

- $\delta < 0$. The non strict inequality is easily proved:

$$\delta = \max_{x \in X', y \in Y \setminus Y'} \{l(x) + l(y) - c\ (x,y)\} \overset{(a)}{\leq} \max_{x \in X, y \in Y} \{l(x) + l(y) - c\ (x,y)\} \overset{(b)}{\leq} 0$$

  (a) is drawn from $X' \times (Y \setminus Y') \subset X \times Y$, and (b) holds since l is a label.
  Now, for the strict inequality: assume that $\delta = 0 \Rightarrow \exists x \in X, y \in Y \setminus Y' : l(x) + l(y) = c(x,y) \Rightarrow (x,y) \in G'$. This implies that x has been visited and is connected to y, but then y should have been visited, so $y \in Y'$, which is a contradiction with the initial assumption. Therefore, $\delta \neq 0$.

- Therefore, when consecutively updating the labels using δ, labels for X will increase, while labels for Y will decrease.

Implementation A pre-coded c++ implementation is used for the solving of the Hungarian Algorithm [20]

3.2.2 Complexity

Equation 24 shows the complexity analysis for the Hungarian Algorithm. The internal part of the *while* loop (3-8) is $O(n^2)$; times reading all n elements of X in the for loop (1); times the amount of times the while loop (2) needs to be iterated. Example 5 shows that the while loop may need to be iterated more than once while trying to find i's assignment. At most, δ will need to be recomputed n times. Therefore, the complexity of this implementation is $O(n^4)$. A faster implementation can be achieved by improving the computation of the augmenting path and δ, by keeping track of them when nodes are added to X', and updating G' only from $Y \setminus Y'$; to a complexity of $O(n^3)$ [21].

```
1  for i ∈ X :                          ← O(n)
2      while i unassigned:              ← O(n)
3          look for augmenting path:    ← O(n²) from Y, ≤ 1 connection; from X, O(n); for n nodes
4          if found: update path        ← O(2n) read the path
5          else: compute δ              ← O(n²) compute l(x) + l(y) − c(x, y) so at most, n × n
6                update label           ← O(2n) read all nodes
7                update matching        ← O(2n) read the path
8                update EG              ← O(n²) check all edges
```
(24)

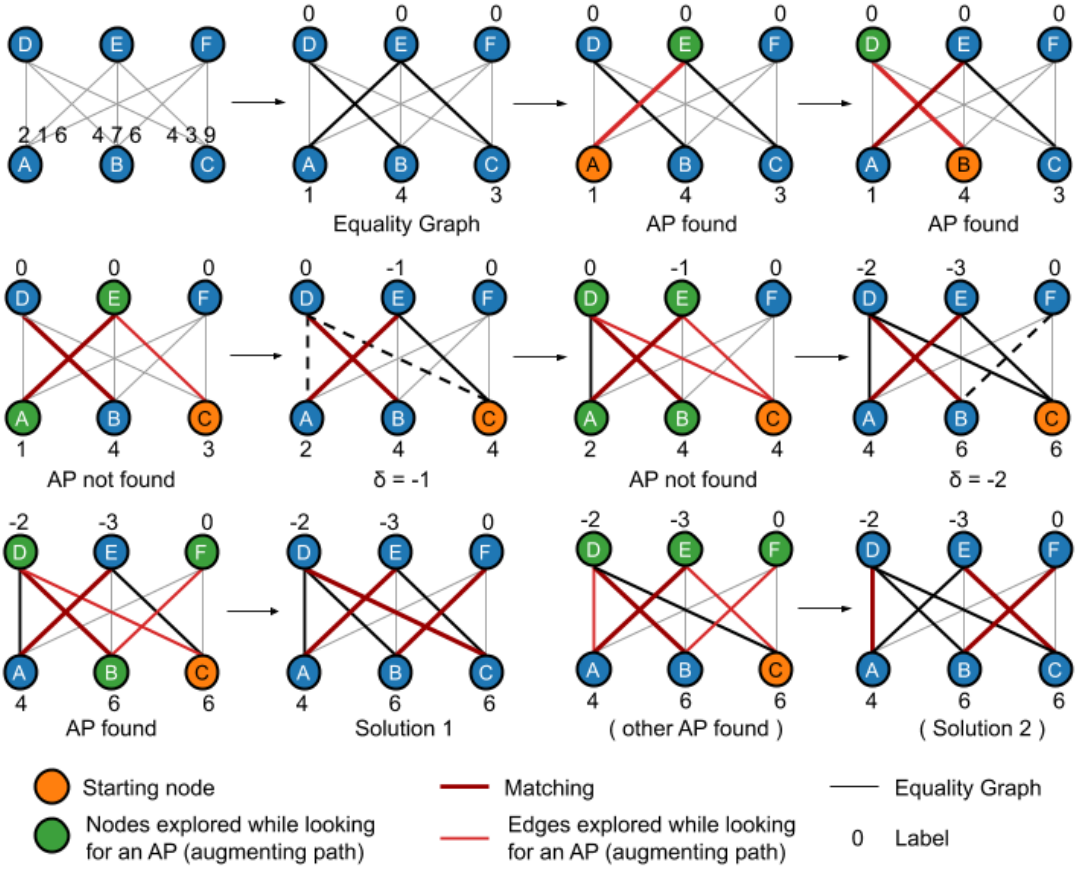

Figure 5: Hungarian Algorithm: computing δ several times per iteration

3.3 Algorithm

3.3.1 General case

The data used considers around $4 \cdot 10^5$ daily requests, which may decrease to $\sim 7 \cdot 10^4$ trips at best for $\alpha = 1.0$, so running the Hungarian algorithm for the whole day is unfeasible and therefore, it must be reduced to smaller assignment problems that can be more easily solved. The resulting match, however, may not be the optimal solution.

Two-step solution

In order to reduce the size of the assignment problem to be solved, it is divided in two main steps: first, the time domain is divided into time intervals of duration Δ and the problem is solved for each one of them; then, a full-day iteration is run, to match those trips that remain unassigned from the first iteration.

Step 1:
Let $[0, T]$ be the time domain and divide it into intervals of duration Δ: : $[0, T] = [\ 0, \Delta\ ] \cup [\ \Delta, 2\Delta\ ] \cup \cdots \cup [\ (d-1)\cdot\Delta, d\cdot\Delta = T\ ]$. For each time interval $[\ \tau\cdot\Delta, (\tau+1)\cdot\Delta\ ]$, the following sets are defined:

$$
\begin{array}{lrcl}
\text{(i)} & T_1 & = & \{t \in \mathscr{T} | \tau\cdot\Delta \leq t.t1 \leq (\tau+1)\cdot\Delta\} \cup \{t \in \mathscr{T} | t \text{ unmatched}, (\tau-1)\cdot\Delta \leq t.t_1 \leq \tau\cdot\Delta\} = \\
 & & & = \{e_i | 1 \leq i \leq m\} \\
\text{(ii)} & T_0 & = & \{t \in \mathscr{T} | \tau\cdot\Delta \leq t.t_0 \leq (\tau+1)\cdot\Delta\} = \\
 & & & = \{s_i | 1 \leq i \leq n\} \\
\text{(iii)} & E & = & \{(e_i, s_j, c) \in T_1 \times T_0 \times \mathbb{R} \text{ such that:} \\
 & & & \text{1. it is feasible:} \quad s_j.s_0 \geq e_i.s_1 + \mathrm{tt}(e_i.s_1, s_j.s_0) \\
 & & & \text{2. its associated cost is non-positive:} \quad c = (1-\beta)\cdot\mathrm{tt}(e_i.s_1, s_j.s_0) - \beta\cdot\delta \leq 0 \\
 & & & \text{where tt represents the travelling\_time function}\} \\
\text{(iv)} & V_1 & = & \{e_1, \cdots, e_m, \widetilde{s_1}, \cdots, \widetilde{s_n} | e_i \in T_1\} \text{ where } \widetilde{s_i} \text{ are virtual nodes} \\
\text{(v)} & V_0 & = & \{s_1, \cdots, s_n, \widetilde{e_1}, \cdots, \widetilde{e_m} | s_i \in T_0\} \text{ where } \widetilde{e_i} \text{ are virtual nodes} \\
\text{(vi)} & \widetilde{E} & = & E \cup \{(\widetilde{s_i}, \widetilde{e_j}, 0) \in T_1 \times T_0 \times \mathbb{R} | \exists c : (e_j, s_i, c) \in E\} \cup \\
 & & & \cup \{(e_i, \widetilde{e_i}, 0) \in T_1 \times T_0 \times \mathbb{R} | 1 \leq i \leq m\} \cup \{(\widetilde{s_i}, s_i, 0) \in T_1 \times T_0 \times \mathbb{R} | 1 \leq i \leq n\}
\end{array}
\tag{25}
$$

The initial bipartite graph (i)-(iii), $G = (T_0 \cup T_1, E)$, defines an unbalanced assignment problem. Moreover, there is no guarantee that there is a solution that matches all edges. Thus, virtual nodes and edges are added to build graph (iv-vi), $\widetilde{G} = (N_0 \cup N_1, \widetilde{E})$, so that the problem is balanced (each side has $n+m$ nodes) and there exists a solution that matches all edges: consider $e_i, \widetilde{s_j} \in V_1$ and $\widetilde{e_i}, s_j \in V_0$:

- those edges from E that were to be matched get assigned within E, and their associated virtual nodes get matched among themselves; i.e., if e_i is matched to s_j, then $\widetilde{s_j}$ is matched to $\widetilde{e_i}$.
- those edges from E that were not to be matched, get assigned to their associated virtual node; i.e., if e_i or s_j are unmatched, then the algorithm matches them to $\widetilde{e_i}$ or $\widetilde{s_j}$, respectively.

The matches computed by applying the Hungarian algorithm to each of the time intervals defined are saved.

Notice that the ending points that are not assigned in a given iteration may be added to the immediate posterior interval. In order to avoid carrying over too many nodes, a condition is added so that each ending node can be considered at most within two intervals, but not more. The nodes that haven't been assigned after two iterations will be explored in the second step.

Step 2:
Consider the set of ending and starting points that remain unassigned after the first step. The edges among them are built as before: taking into account the feasibility condition and keeping only those whose cost is non-positive (since they are the only ones that can be selected, as proved in section *Interpretation of δ and other properties*), to reduce the memory being used.

The Hungarian algorithm is then run, to check if they can be assigned to trips outside of their previous time interval, and the matchings obtained from it are added to the previously computed ones.

Note: this procedure does not reach an optimal assignment.
Note: for the following algorithm, the functions *balance_ trips* and *Hungarian_ algorithm* are used to simplify the representation: *balance_ trips* returns $V_1, V_0, \widetilde{E}$ as defined in (iv)-(vi) given the sets T_1, T_0, E, while *Hungarian_ algorithm* uses the previous output to return the set of edges used in the assignment.

Algorithm 3: Rebalancing: general case

Input : Ridesharing output: set of trips $\mathscr{T} = \{t | t = \{id, s_0, s_1, t_0, t_1\}\}$
Output: File with assignments saved for each station

// Load trips, travelling times matrix and parameters
$\mathscr{T} \leftarrow$ Ridesharing output
tt $\leftarrow$ Travelling times matrix
T $\leftarrow$ Time limit
$\Delta \leftarrow$ Time interval duration

// Initialize variables
$T_1 \leftarrow \emptyset$
$T_2 \leftarrow \emptyset$
$\tau \leftarrow 0$
$M \leftarrow \emptyset$

// Iterate over time slots
while $\tau \cdot \Delta < T$ **do**
    // Get nodes
    $T_1 \leftarrow \{e \in \mathscr{T} | \tau \cdot \Delta \leq e.t_1 \leq (\tau + 1) \cdot \Delta\} \cup T_1$
    $T_0 \leftarrow \{s \in \mathscr{T} | \tau \cdot \Delta \leq s.t_0 \leq (\tau + 1) \cdot \Delta\}$

    // Get edges
    $E \leftarrow \{(e_i, s_j, c) \in T_1 \times T_0 \times \mathbb{R} | s_j.t_0 \geq e_i.t_1 + \text{tt}(e_i.s_1, s_j.s_0), c = (1-\beta) \cdot \text{tt}(e_i.s_1, s_j.s_0) - \beta \cdot \delta \leq 0\}$

    // Balanced bipartite graph
    $V_1, V_0, \widetilde{E} \leftarrow$ balance_trips(T_1, T_0, E)

    // Apply Hungarian algorithm
    $match \leftarrow$ Hungarian_algorithm$(N_1, N_2, \widetilde{E})$

    // Save match and prepare for next iteration
    for $(e_i, s_j, c) \in E$ **do**
        if $(e_i, s_j) \in match$ **then**
            $M \leftarrow M \cup \{(e_i, s_j)\}$
            $T_1 \leftarrow T_1 \setminus \{(e_i, s_j)\}$
        end
    end
end

// Iterate over full day
$T_1 \leftarrow \{e \in \mathscr{T} | \nexists s : (e, s) \in M\}$
$T_0 \leftarrow \{s \in \mathscr{T} | \nexists e : (e, s) \in M\}$
$E \leftarrow \{(e_i, s_j, c) \in T_1 \times T_0 \times \mathbb{R} | s_j.t_0 \geq e_i.t_1 + \text{tt}(e_i.s_1, s_j.s_0), c = (1 - \beta) \cdot \text{tt}(e_i.s_1, s_j.s_0) - \beta \cdot \delta \leq 0\}$
$V_1, V_0, \widetilde{E} \leftarrow$ balance_trips(T_1, T_0, E)
$match \leftarrow$ Hungarian_algorithm$(N_1, N_2, \widetilde{E})$
for $(e_i, s_j, c) \in E$ **do**
    if $(e_i, s_j) \in match$ **then**
        $M \leftarrow M \cup \{(e_i, s_j)\}$
    end
end

print(M) to file

3.3.2 Case β=0

Algorithm

As seen in section 3.1.3, for $\beta = 0$, a faster solution can be achieved by taking into account that the only possible assignments allowed are those within the same station.

Let $\mathscr{S}$ be the set of stations. For $\beta = 0$ and $a, b \in \mathscr{T}$; an assignment $a \to b$ is feasible if: $a.t_1 \leq b.t_0$, $a.s_1 = b.s_0$, since it is enough to solve the problem individually for each station:

$$\text{argmax}_{x_{ij}} \left\{ \sum_{ij} x_{i,j} \middle| i.s_1 = j.s_0 \right\} = \sum_{s \in \mathscr{S}} \text{argmax}_{x_{ij}} \left\{ \sum_{ij} x_{i,j} \middle| i.s_1 = j.s_0 = s \right\} \tag{26}$$

Let $S_s = \{t \in \mathscr{T} | t.s_0 = s \in \mathscr{S}\}$, the set of trips that start at station s, ordered by increasing starting time: $S_s = \{s_1, s_2, \cdots, s_n\}$ where $s_1.t_0 \leq s_2.t_0 \leq \cdots \leq s_n.t_0$. Similarly, let $E_s = \{t \in \mathscr{T} | t.s_1 = s \in \mathscr{S}\}$, the set of trips that end at station s, ordered by increasing ending time: $E_s = \{e_1, e_2, \cdots, e_m\}$ where $e_1.t_1 \leq e_2.t_1 \leq \cdots \leq e_m.t_1$.

Algorithm 4 reads the set E_s and assigns, for each ending point, the first element of S_s that is feasible and is yet unassigned. Therefore, the assignment function is given by:

$$\begin{array}{l} \forall s \in \mathscr{S}, \\ \quad \begin{array}{rcl} M_s : E_s & \longrightarrow & S_s \\ e_i & \longmapsto & s_j \text{ with } j = \mathscr{M}(s)(e_i) \end{array} \\ \text{where:} \\ \quad \begin{array}{rcl} \mathscr{M}(s) : E_s & \longrightarrow & \{1, 2, \cdots, |S_s|\} \\ e_1 & \longmapsto & \min_{1 \leq i \leq |S_s|} \{i | e_0.t_1 \leq s_i.t_0, s_i \in S_s\} \\ e_j & \longmapsto & \min_{1 \leq i \leq |S_s|} \{i | e_j.t_1 \leq s_i.t_0, s_i \in S_s, i > \mathscr{M}(s)(e_{j-1})\}, \forall j > 1 \end{array} \end{array} \tag{27}$$

Optimal assignment *Function 27 yields an optimal solution for problem 26. In fact, given the ordered sets E_s and S_s, with cardinals m and n respectively, the algorithm maximizes the number of connections between E_s and the first p elements of S_s, $S_s^p = \{s_1, \cdots, s_p\}, \forall p : 1 \leq p \leq n$.*

Proof. Let's first consider some simple properties, and then prove the statement by induction:

If assignment $e_i \in E_s \to s_j \in S_s$ is feasible, then:	If assignment $e_i \in E_s \to s_j \in S_s$ is not feasible, then:
(P.i) assignment $e_i \to s_{j+1}$ is also feasible	(P.iii) assignment $e_i \to s_{j-1}$ is also unfeasible
(P.ii) assignment $e_{i-1} \to s_j$ is also feasible	(P.iv) assignment $e_{i+1} \to s_j$ is also unfeasible

- Base case: $p = 1 \Rightarrow S_s^1 = \{s_1\}$.
  If $\exists e_i \to s_0$, a feasible assignment, by applying (P.ii), $e_0 \to s_0$ is also feasible.
  Otherwise, if $\forall i, e_i \to s_0$ is unfeasible, then, in particular, $e_0 \to s_0$ is also non feasible.
- Let's assume it true for all natural numbers up to p, and let's prove it for $p+1$. Let $k(i)$ be the maximum number of connections up to node i.
  - If $k(p) = k(p+1)$, it is done, since a solution with $k(p)$ connections has already been found.
  - Otherwise, $k(p+1) = k(p) + 1 = K + 1$. Let's see that an optimal assignment with $K+1$ connections can be build with the one found for p nodes plus the feasible edge $e_{K+1} \to s_{p+1}$.
    If e_K is not assigned to $s_p \implies e_{K+1} \to s_p$ is unfeasible $\implies$ by (P.iii) $e_{K+1} \to s_i$ is unfeasible $\forall i \leq p \implies$ by (P.iv), $e_j \to s_i$ is unfeasible $\forall i \leq p$, $j \geq K+1$. Therefore, if the addition of s_{p+1} yields an extra connections, then this match must be: $e_j \to s_{p+1}$ with $j \geq K+1$ (otherwise, an earlier $s_{\leq p}$ connection would be lost) $\implies \exists j \geq K+1 : e_j \to s_{p+1}$ is feasible $\implies$ by (P.ii), $e_{K+1} \to s_{p+1}$ is feasible.
    If e_K is assigned to s_p, let's consider two different cases. If $e_{K+1} \to s_p$ is unfeasible, then the reasoning is the same as in the previous case. If otherwise, $e_{K+1} \to s_p$ is feasible $\implies$ by (P.i) $e_{K+1} \to s_{p+1}$ is feasible.

□

Algorithm 4: Rebalancing for $\beta = 0$

Input : Ridesharing output: set of trips $\mathscr{T} = \{t | t = \{id, s_0, s_1, t_0, t_1\}\}$
Output: File with assignments/matchings saved for each station

```
// Load trips
```
$\mathscr{T} \leftarrow$ Ridesharing output

```
// Solve problem for each station
```
for $s \in \mathscr{S}$ **do**
```
    // Get trips starting/ending at d
```
$S_s \leftarrow \{t \in \mathscr{T} | t.s_0 = s\}$
$E_s \leftarrow \{t \in \mathscr{T} | t.s_1 = s\}$

```
    // Order S_s by increasing t_0 and E_s by increasing t_1
```
$S_s \leftarrow$ merge_sort (S_s, t_0)
$E_s \leftarrow$ merge_sort (E_s, t_1)

```
    // Create vector to save results
```
$R_s \leftarrow$ Vector(size: $|E_s|$, init: -1)

```
    // Begin assignment
```
$i = 0$
$j = 0$
while $i < |E_s|$ ***and*** $j < |S_s|$ **do**
```
        // Assign trip i to trip j
```
    if $E_s[\ i\ ].t_1 <= S_s[\ i\ ].t_0$ **then**
        $R[\ i\ ] = j$
        $++i$
    end
    $++j$
end
print(R_s) to file
end

Comparability

For $\beta = 0$, an optimal assignment can be obtained by applying algorithm 4, as seen in the previous section. However, for $\beta \neq 0$, the assignment is only optimal for some time intervals.

The algorithm for $\beta = 0$ can be adjusted to match the procedure of the general case, dividing the time domain in several time slots, solving the problem first for each of those intervals, and then solving the problem for those trips that remain unassigned, as shown in algorithm 5.

Algorithm 5: Rebalancing for $\beta = 0$, with time slots

Input : Ridesharing output: set of trips $\mathscr{T} = \{t | t = \{id, s_0, s_1, t_0, t_1\}\}$
Output: File with assignments/matchings saved for each station

$\mathscr{T} \leftarrow$ Ridesharing output, $T \leftarrow$ Time limit, $\Delta \leftarrow$ Time interval duration
for $\tau \in (0, T/\Delta)$ **do**
    trips' $\leftarrow \{t \in \mathscr{T} | t$ unassigned$[(\tau - 1) \cdot \Delta \leq t.t_1 \leq (\tau + 1) \cdot \tau || \tau \cdot \Delta \leq t.t_0 \leq (\tau + 1) \cdot \tau]\}$
    Algorithm 4 (trips')
end
for $t \in \mathscr{T}$ **do**
    trips' $\leftarrow \{t \in \mathscr{T} | t$ unassigned$\}$
    Algorithm 4 (trips')
end

4 Data

4.1 Spatial data: Manhattan

4.1.1 Graph

The set of trips to be used as a basis for applying the ridesharing and rebalancing algorithms take place in Manhattan. In order to describe these trips, the space needs to be mapped and some sets need to be defined: the roads on which the vehicles can circulate and the direction of traffic on said streets, the spots where passengers can be picked up from or dropped off to, and the paths between those points, with the corresponding travelling times and distances covered.

Graph definition The streets and pathways can be modelled as a directed graph $G = (S, E)$ in which the edges $e = (s_0 \to s_1) \in E,\ s_0, s_1 \in S$ represent sections of roads [22]:

1. on which the circulation is allowed in the given direction,
2. on which the circulation speed is assumed to be constant (both within the edge and along the day),
3. that do not intersect other edges (a new node on the crossroads would be defined otherwise),
4. that are assumed to be straight lines for the graphical representations.

While a more exact approximation of the real roads could be achieved by adding more nodes to better represent curves, all cost functions are defined with travelling times, so the representative data is actually the time matrix (section 4.1.2) and the topology of the graph, i.e., how the nodes are connected and relate to each other.

The resulting graph, in which the nodes are represented on its corresponding coordinates, can be found on figure 6. The distinction between one-way and two-way streets is made, and those cases in which the circulation is only allowed in one direction are divided on roads heading East and roads heading West.

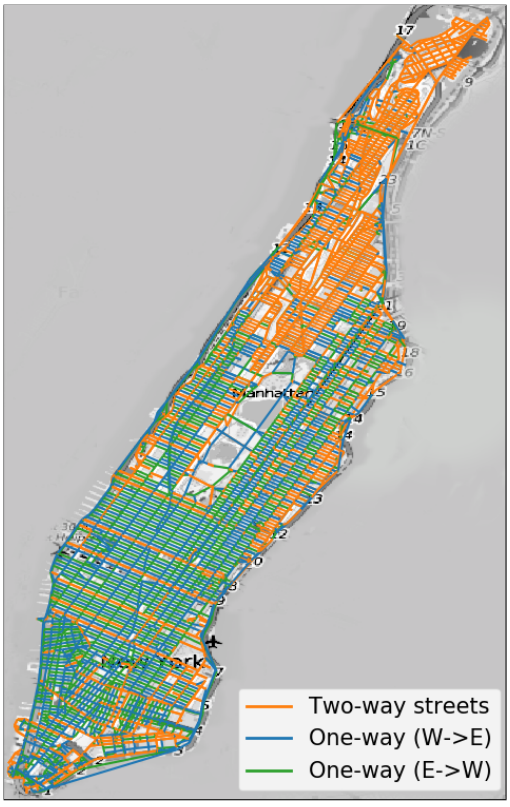

Figure 6: Direction of traffic

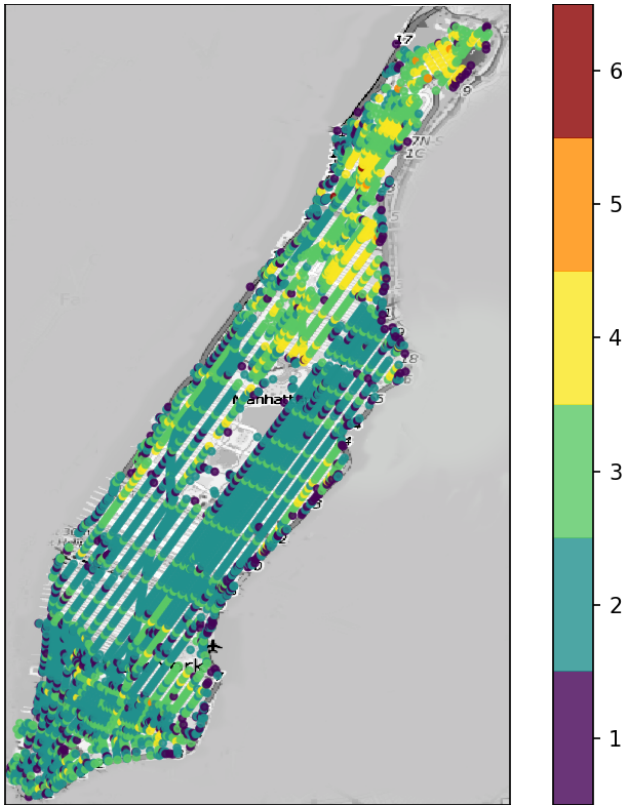

Figure 7: Outdegree of nodes

Graph characteristics The graph $G = (S, E)$ is a directed, simple (there are no loops - edges starting and ending on the same node - or multiple adjacencies - for every two different nodes, there is at most one edge connecting them in a given direction), strongly connected (for every two different nodes, exists a directed path that goes from one to another) and planar (there exists an embedding of the graph into the Euclidean plane, so that nodes and edges can be represented in a classical plane surface as points and lines, without intersections between lines and without nodes overlapping on lines they do not belong to) graph.

Let $S = \{s_1, \cdots, s_n\}$ be the set of nodes, let $E = \{e_1, \cdots, e_m\}, m \leq (n-1)^2$ be the set of edges and let $e_k = (s_i, s_j) = e_{(i,j)}$ denote the arrow from s_i to s_j: $s_i \to s_j,\ s_i, s_j \in S$. The graph $G = (S, E)$ defined has $|S| = 4091$ nodes and $|E| = 9452$ edges.

The indegree $d-$ and outdegree $d+$ of a given vertex is defined as:

$$\begin{array}{rccl} d-: & S & \longrightarrow & \mathrm{N} \\ & s_i & \longmapsto & |\{e_{(j,i)\in E}\}| \end{array} \qquad \begin{array}{rccl} d+: & S & \longrightarrow & \mathrm{N} \\ & s_i & \longmapsto & |\{e_{(i,j)\in E}\}| \end{array} \tag{28}$$

The indegree of a node represents the number of edges that end at the node, and is the sum of the elements of that node's column on the adjacency matrix, a $|S| \times |S|$ square matrix in which the element at row i and column j represents the number of edges that start from s_i and end at s_j (since the graph is simple, the matrix will be hollow and only contain zeros and ones; since the graph is directed, it may not be symmetrical). Similarly, the outdegree of a node represents the number of edges that start at the node, and is the sum of the elements of that node's row on the adjacency matrix. Since the graph is strongly connected, at least one element of each row and column of the adjacency matrix will be non-zero, and therefore, $d+(s) > 0$ and $d-(s) > 0,\ \forall s \in S$.

The average indegree and outdegree of the nodes of the graph is $|E|/|S| \simeq 2.31$. Both the maximum indegree and maximum outdegree in the graph is 6. In fact, the indegree and outdegree distributions are quite similar; out of the 4091 nodes:

Possible values	0	1	2	3	4	5	6
Indegree	0	468	2210	1097	308	7	1
Outdegree	0	463	2211	1113	294	8	2

Table 6: Number of nodes whose indegree and outdegree take a certain value

More than two thirds of the nodes have the same value of indegree as outdegree. In fact, the percentage of nodes that have indegree equal to outdegree out of all the nodes that have a given indegree is higher than 70% for all cases except for those with indegree 1 ($\sim 21\%$). The first case corresponds to intersections of streets that do not end on that corner, while the second case corresponds to roads with no intersections: these are mainly dead-end streets (all two-way streets), while a few are corners around blocks.

The fact that 2 is the mode for both in and outdegrees can be justified in the same way: they are intersections of two unidirectional streets. An example of this is the perpendicular grid of one-way roads on the southern part of Manhattan: one street goes in one direction while the parallel ones follow the opposite direction; the orange lines that go across (figure 6) correspond to nodes with outdegree 3 (figure 7), on which one of the streets is bigger and traffic is allowed on both directions. Another example is the northern part, since most streets are two-way, the in and outdegrees are higher.

The outdegree of the nodes of the graph is represented on figure 7. Since there are not big distances between connected nodes, the in and outdegrees are small compared to the total amount of nodes (sparse graph), and most nodes have the same in and outdegrees, the graphical representation of the indegrees is very similar to the outdegrees one, and thus, is not presented.

Nodes and stations For the ridesharing algorithm, it is assumed that trips can only begin and end at certain locations, called stations, and taxis can only appear at or be produced from these stations. In practice, this would mean that a person could only catch a taxi at the intersection of at least two streets, a corner or the end of a road.

4.1.2 Time matrix

Definition The time matrix *times* is a $|S| \times |S|$ square hollow matrix in which the element at row i and column j is non-negative and represents the time it takes to travel from node s_i to node s_j. Each travelling time is rounded to the superior integer in seconds; this guarantees that all elements outside the diagonal are different from zero. Since the graph is directed, the time matrix may not be symmetrical.

Computing times Depending on if the two nodes are connected or if a path between them must be found (which is guaranteed to exist since the matrix is strongly connected), the computation method changes:

- **Time between connected nodes**: the time it takes to travel a street may vary from day to day and within the day, depending on factors as the time, traffic congestion or momentary weather conditions. As a simplification, it is taken as the average of the time it takes to travel along the edge throughout the day (data available for each one-hour slot, with a mean standard deviation of 12.8 seconds).
- **Time between unconnected nodes**: a feasible path between two unconnected nodes, i.e. an ordered series of pairwise connected nodes starting from the origin node and ending at the destination node, must take into account the direction of the edges. For non trivial paths (for paths whose starting and ending nodes are different), the shortest path problem for directed weighted graphs must be solved. In this case, the Dijkstra Algorithm is applied to graph $G = (S, E)$ with the costs associated to the edges computed in the previous step. This also allows to generate a paths file, in which the sequence of stations to go through from one station to another are recorded.

Usage While these paths file would be useful when computing the location of a vehicle given its origin, destination and starting time of the trip, it takes up considerable memory space. This is why, instead of using the paths file, each path is recomputed using the times matrix.

Let n be the number of stations, let $times[s_i \to s_j]$ be the time it takes to go from s_i to s_j, and let path $P_{s_i \to s_j}$ be the path that connects s_i with s_j, then:

$$times[s_i \to s_j] = times[s_i \to s] + times[s \to s_j],\ \forall s \in P_{s_i \to s_j} \tag{29}$$

$$P_{s_i \to s_j} = \{s_i, s_j\} \iff \nexists s_k \in \{s_l\}_{l=1}^{n} \setminus \{s_i, s_j\} : times[s_i \to s_j] = times[s_i \to s_k] + times[s_k \to s_j] \tag{30}$$

Algorithm 6: Computing paths from time matrix

Input : Times matrix, stations s_i, s_j.
Output: $P_{s_i \to s_j}$

$n \leftarrow \text{length}(times)$
$s_{aux} \leftarrow s_i$
$s_{min} \leftarrow s_j$
$P_{s_i \to s_j} \leftarrow \{s_i\}$
while $s_{aux} \neq s_j$ **do**
  for $k \in \{1, 2, \cdots, n\}$ **do**
    if $times[s_{aux} \to s_j] = times[s_{aux} \to s_k] + times[s_k \to s_j]$ ***and*** $s_k \neq s_{aux}$ **then**
      if $times[s_{aux} \to s_k] < times[s_{aux} \to s_{min}]$ **then**
        $s_{min} = s_k$
      end
    end
  end
  $P_{s_1 \to s_2} \leftarrow P_{s_1 \to s_2} \cup \{s_{aux}\}$
  $s_{aux} \leftarrow s_{min}$
  $s_{min} \leftarrow s_2$
end

Statistics The average time it takes to travel between two stations is approximately 1132 seconds (18' 52"). The histogram of the different time travels is shown in figure 8.

The time matrix is not symmetrical. The histogram of the difference between travelling from s_i to s_j and travelling from s_j and s_i is shown in figure 9. This difference matrix is skew-symmetrical, and thus, the sum of its elements equals zero. Its fitting to a Gaussian distribution yields an average of 0 and a standard deviation of 138. By repeating the process for only those stations that are directly connected, figure 9 is obtained.

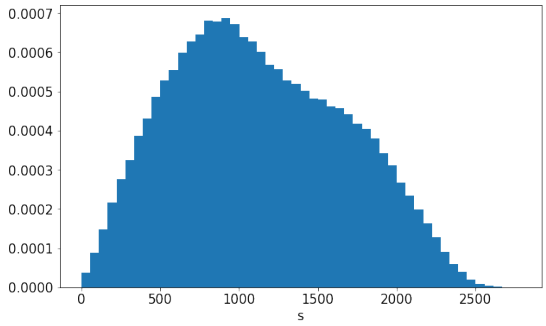

Figure 8: Histogram of times matrix

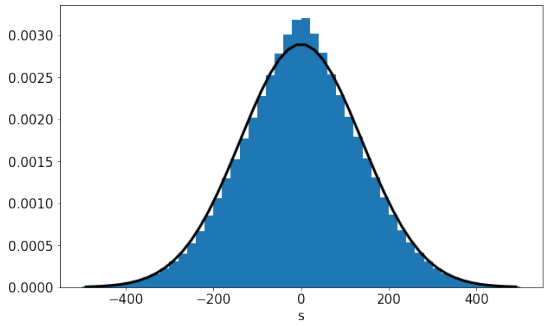

Figure 9: $times[s_i \to s_j]$ - $times[s_j \to s_i]$

Since the time matrix is non symmetrical and one-way streets exist, the path to go from s_i to s_j may not be the path to go from s_j to s_i travelled in the opposite direction. This is shown on figure 10: the trip (from the filled red dot to the empty red dot) in the first case takes 731 seconds, while the second case (trip on the opposite direction) takes 712 seconds. The distance travelled varies too: the real distance on the sphere between the two points is 1.86km. The first case shows how algorithm 6 may yield several different paths with the same cost, for a certain starting and ending point pair. The orange path is 2.46km long, the blue one is 2.49km long, while the green one measures 3.35km.

Figure 10: Comparison between $times[s_i \to s_j]$ and $times[s_j \to s_i]$

4.1.3 Distance matrix

Definition The distance matrix is a $|S| \times |S|$ square matrix in which the element at row i and column j represents the length of the fastest path between node s_i and s_j. All elements must follow the properties of the distance function, and thus, apart from being hollow and having no negative elements, the triangular inequality must hold. The distance matrix, like the times matrix, is not symmetrical.

Computation The distance matrix can be computed while computing the times matrix: once the Dijkstra Algorithm has found the time-wise shortest path between two stations ($s_1 = p_0, p_1, \cdots, p_n, s_2 = p_{n+1}$), the distance between them is the sum of the distance of each step $\sum_{i=0}^{n} d(p_i, p_{i+1})$.

Distance computation: Euclidean vs Haversine distance It is assumed that edges are actually straight lines between two nodes. Computing the distance between two unconnected nodes means adding the distances of each of the segments of its path as defined above. Computing the distance between two connected edges means calculating the distance between two points given by their coordinates.

However, depending on which distance is being used, the results may vary. If the points are assumed to be on a spherical surface as Earth's is approximately, then the distance between the points must be computed as the distance along the geodesic curve that contains them both - in a sphere, a great-circle - it will be and the coordinates are to be taken as a longitude, latitude pair. This distance is given by the haversine formula [23]:

$$d_h = 2 \cdot r \cdot \arcsin \sqrt{\sin^2\left(\frac{y_2 - y_1}{2}\right) + \cos y_1 \cdot \cos y_2 \cdot \sin^2\left(\frac{x_2 - x_1}{2}\right)} \tag{31}$$

where r is Earth's radius, which is taken to be 6367km, x_1, x_2 are the longitudes and y_1, y_2 are the latitudes.

If the points are assumed to be on a plane, then the Euclidean distance is used: $d((x_1, y_1), (x_2, y_2)) = \sqrt{(x1 - x2)^2 + (y1 - y2)^2}$. The coordinates could be "transformed" into kilometers by applying the conversion rate 1 hexadecimal degree equates 111.13km, which is true for Earth's equator. However, this is a poor approach. The distance along the y-axis can be better approximated by computing the length of the arc of circumference with radius equal the Earth's and the angle between both values of the latitude. The distance along the x-axis can be better approximated by applying a conversion rate adjusted to the circumference on the horizontal plane at the center of the map, or at the average of latitudes. Assuming $\{(x_i, y_i)\}_{i=\{1,2\}}$ are the longitude-latitude coordinates of the points, and that y_0 represents the latitude of the middle point of the map:

$$d_{e1} = r\frac{2\pi}{360}\sqrt{\left(\cos\left(y_0\frac{2\pi}{360}\right)(x_2 - x_1)\right)^2 + (y_2 - y_1)^2} \tag{32}$$

$$d_{e2} = r\frac{2\pi}{360}\sqrt{\left(\cos\left(\frac{y_1 + y_2}{2}\frac{2\pi}{360}\right)(x_2 - x_1)\right)^2 + (y_2 - y_1)^2} \tag{33}$$

Figure 11 shows, given several points along straight lines on the map, the real distance (in meters) between those points and the down left corner one computed with the haversine formula and the absolute error (in meters) made when computing the distance with Euclidean approximations. The distance used in the simulations is haversine's.

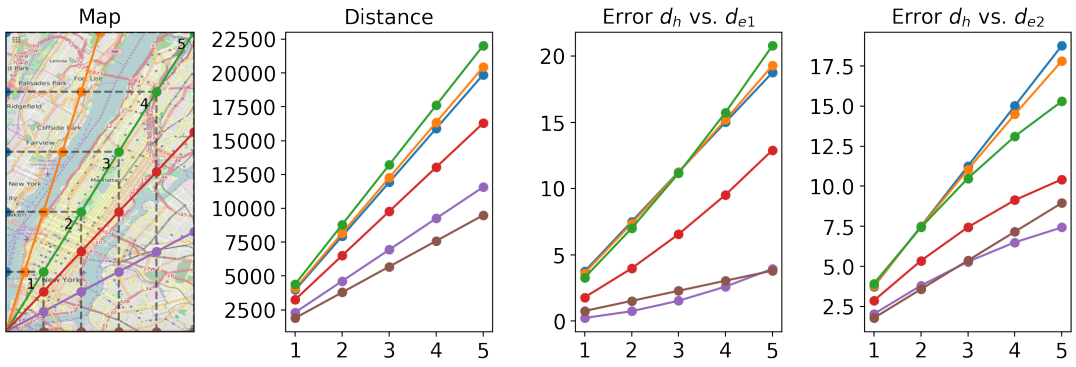

Figure 11: Distance between points in the map

4.2 Demand

4.2.1 Data source and processing

Source The demand data used contains requests attended by taxis at Manhattan, from the eighteenth to the twenty-first week of the year 2013. The coordinates and timestamps of pick-up and drop-off points, the number of passengers and the length and duration of the each trip are provided.

Discretization The demand data consists on sets of pairs of coordinates and timestamps for pick-up and drop-off points for each of the trips carried out on a given day. However, as stated in section 4.1.1, only a set of 4091 points can be starting or ending points of trips. By assigning each pair of coordinates to the station closest to it, the demand data gets spatially discretized. The "influence" area of each of the stations is given by the Voronoi polygon defined by the set of stations,as shown on figure 12.

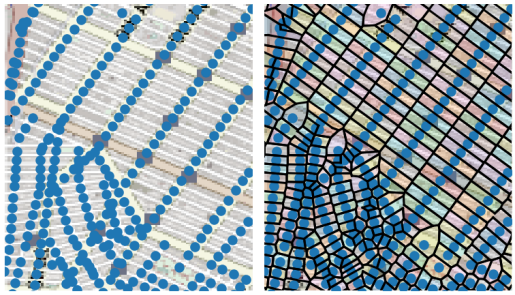

Figure 12: Stations and influence area

4.2.2 Demand analysis

Length and distance distributions The demand data is the baseline upon which the ridesharing and rebalancing are run; therefore, the demand data gives an upper bound to the amount of vehicles to be used. The effect of the different parameters α (ridesharing) and β (rebalancing) will later be analyzed on these two quantities. Taking one of the days as an example (week 19, day 2), the length and duration of its 421582 trips is computed. Only 162 trips last longer than an hour (less than 0.04%) and only one of them lasts longer than two hours; these values are considered outlayers and aren't shown on the histogram in figure 13. Similarly, only seven trips longer than 20km are deleted from the histogram in figure 14. The main statistics of these two distributions can be found on table 7.

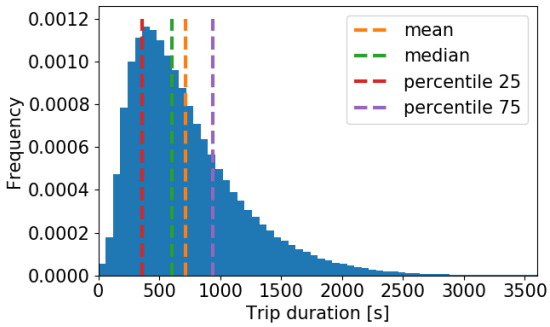

Figure 13: Histogram: trips by duration

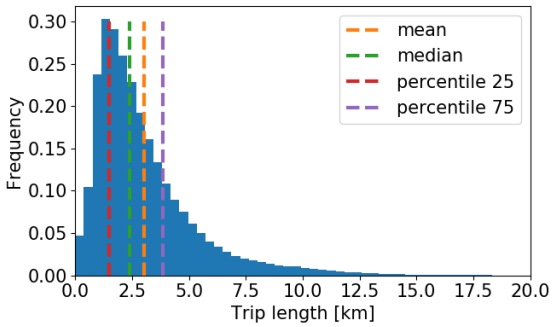

Figure 14: Histogram: trips by length

	Duration[s]	Length[km]
Mean	712.5	3.02
Std	469.0	2.27
Mode	360	0*
Max	17950	22.90

	Duration[s]	Length[km]
Percentile 25	360	1.48
Percentile 50	600	2.39
Percentile 75	940	3.84
Percentile 95	1620	7.65

Table 7: Statistics from trip length and duration distributions

Geographical distribution *NYC Open Data* [24] provides open source data of the coordinates that define 65 different zones in the borough of Manhattan [25], roughly based on NYC Department of City Planning's NTAs (Neighborhood Tabulation Areas). The demand trends change among the different zones: for example, in figure 15, it is shown the distribution of pick-up and drop-off demand points on a given day. While the northern part of Manhattan shows a small demand (in the northern half of the map, only a 4.9% of the total pick-up demands take place, which decreases to 0.3% if only the northern third of the map is considered; respectively, 6.6% and 0.9% for drop-off), the southern regions gather a significant share of the global demand (the three zones with higher pick-up demand - Upper East Side South and North, and Midtown Center - contain a 13.4% of the demand, which increases to a 21.6% if the next two top zones are added - Midtown East and Union Square. The data for drop-off demands yields similar results, with 14.4% of the demand in the same three top positions, and 21.6% when adding the next two top zones - Midtown East and Murray Hill).

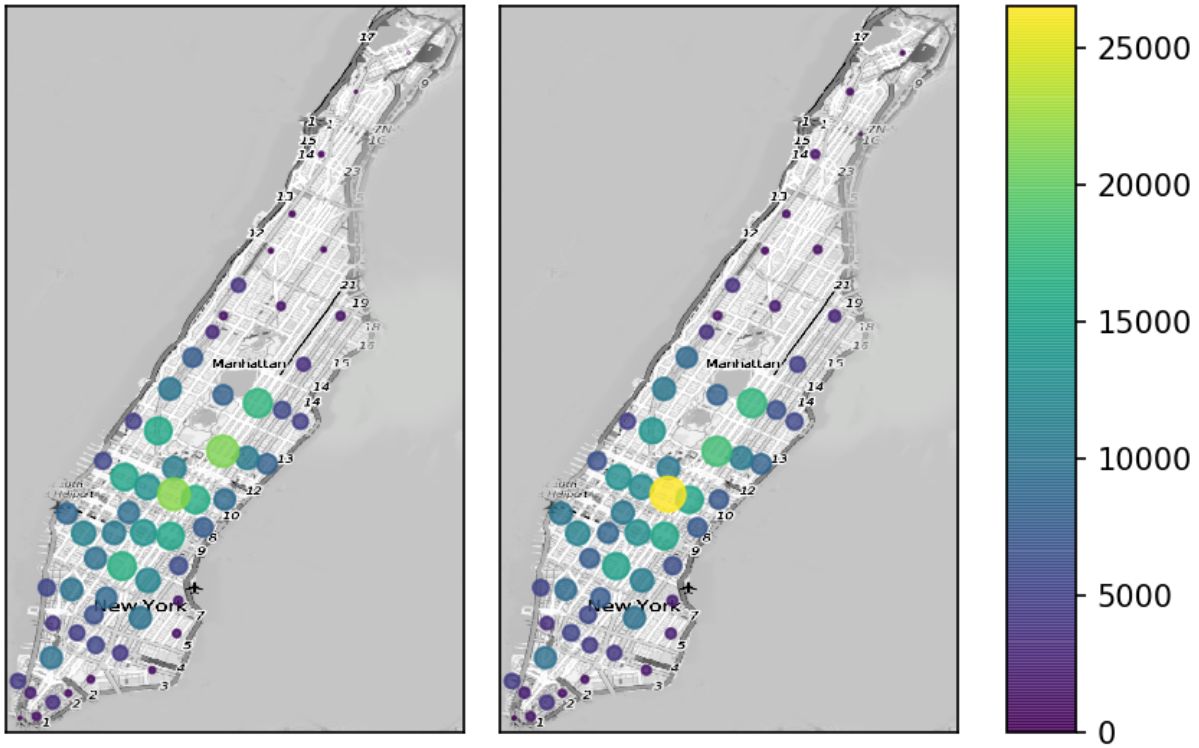

Figure 15: Demand data over full day for pick-up (left) and drop-off (right) points

Time distribution The data contains trips that extend over a one-month period. The demand trends vary temporally in two main ways: among days (weekday vs. weekend) and within the same day (rush-hour traffic is identifiable), as shown in figure 16; for example, weekends yield a smaller total number of trips and present a delayed curved, with peak hours starting later and lasting well into the night. The geographical distribution also varies with time, as is the case for weekdays, which show a very focused set of destination points during morning rush-hour and a more uniform distribution during the afternoon and evening, as shown on figures 17-18.

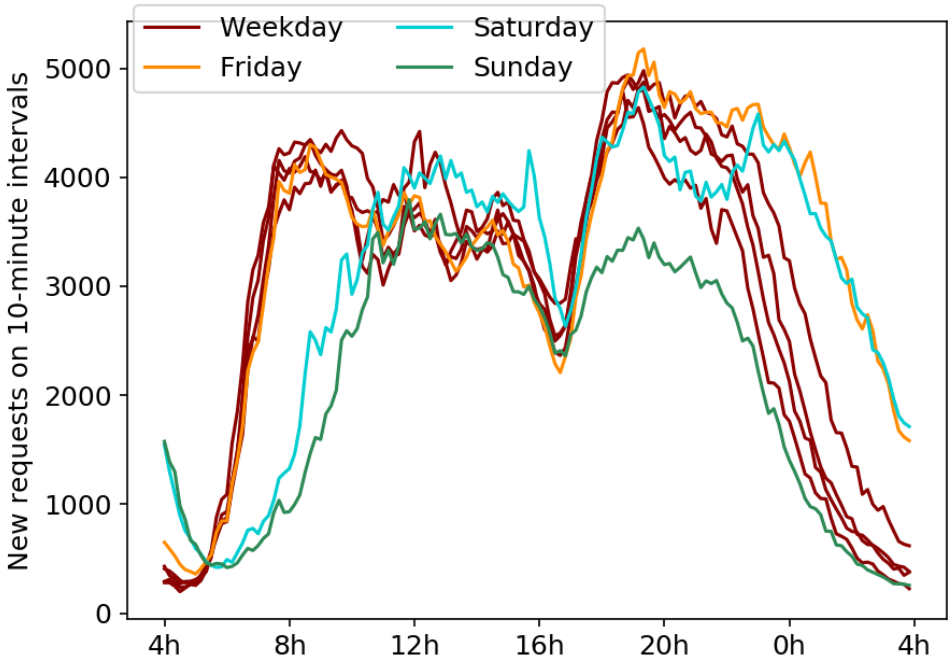

Figure 16: Demand per time and day

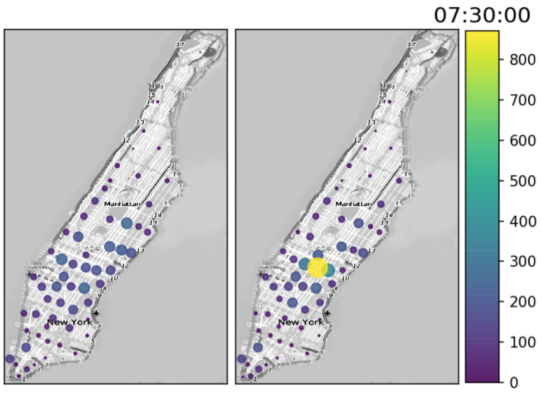

Figure 17: Demand at Wednesday 7:30

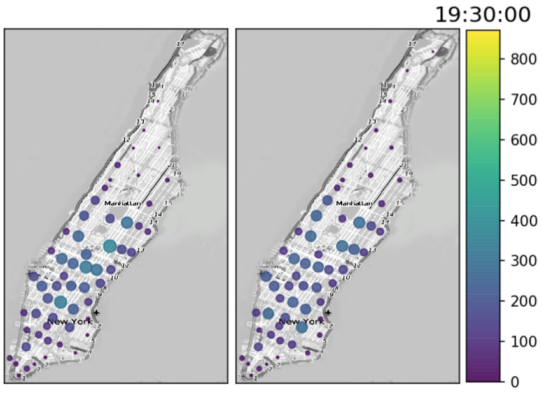

Figure 18: Demand at Wednesday 19:30

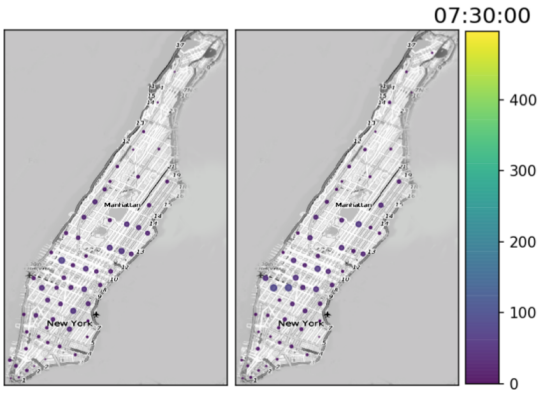

Figure 19: Demand at Saturday 7:30

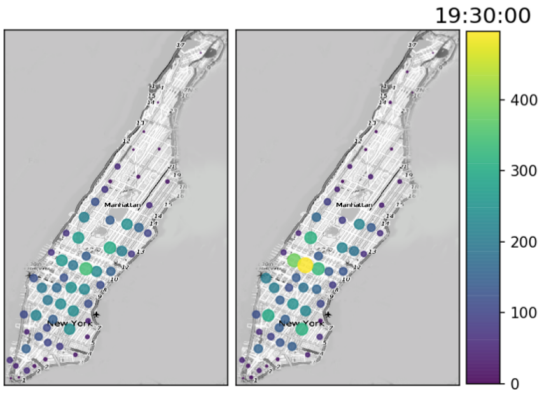

Figure 20: Demand at Saturday 19:30

5 Results

5.1 Implementation

5.1.1 Machines

Depending on the section and set of parameters being ran, different memory usages are needed; for example, up to $\sim 17GB$ for running the rebalancing code for $\alpha = 0.0, \beta = 1.0, \Delta = 1h$. This, added to the amount of simulations attempted and the duration of running each one of them, leaded to the need to run those simulation in AWS (Amazon Web Services). The instance type is chosen depending on the code and parameters being run: while rebalancing for $\alpha = 0.0, \beta = 1.0, 0.8$ uses memory-optimized instances, the pooling algorithm does not require as much memory, and can be run in a normal one.

5.1.2 Output format

The output has been summarized in several files, so that, for each day and α parameter, the results from the pooling simulation use around 350MB of memory. By adding the results for rebalancing with $\beta = 0, 0.2, \cdots, 1$, extra 75MB of memory are used. These results may vary depending on the number of requests made during the day or the parameter being considered.

Ridesharing	
For each trip	Its identification number
	Its starting and ending time and station
	Its position at the start of every time-step
	The total number of passengers carried
	The order in which those passengers are picked-up and dropped-off
	The total distance travelled
For each passenger	Its identification number
	The identification number of the trip that contains it
	The time of the request
	Its starting and ending time and station
Overall	Distribution of number of vehicles per capacity usage for every time-step
	Summary of parameters employed

Rebalancing	
For each vehicle	The identification number of both trips involved
	The time and station at which the first trip ends
	The time and station at which the second trip starts
	The time it actually arrives to the second stations (defines waiting time)
	The total distance travelled while rebalancing
For each iteration	Number of nodes on each side of the bipartite graph
	Number of edges considered
	Number of edges disregarded due to unfeasibility
	Number of resulting assignments
	Time it takes to solve the Hungarian Algorithm

Table 8: Output description

Note: the position of all vehicles in all moments of time may not be completely recoverable. If a vehicle has a change in its plan during its execution that makes it take a detour from the final plan's optimal path, it will show in the fact that the trip lasts longer than it should. The exact position may not be recovered with absolute certainty, but a very reliable approximation can be obtained by comparing the ideal path of the final plan, each of its passengers' pick up and drop off times, and the position of the vehicle at the start of every 30 second time-step.

5.2 Approach

Simulations are computed for 7 days, for several parameters: ridesharing is run for $\alpha \in \{0.0, 0.2, 0.4, 0.6, 0.8, 1.0\}$, and the rebalancing parameter β is run for the same values, with $\delta = 600$ and the corresponding maximum relocating times of $(0s, 150s, 400s, 900s, 2400s, \infty)$. Therefore, for every day of demand data, 36 possible scenarios are generated, besides the baseline of one vehicle per request.

These two parameters have different effects on the final result: for example, while the total distance travelled by the fleet depends on both α and β, the delay of each of the passengers depends only on α and is completely independent of the value of β; however, the time spent idling is only determined by parameter β.

5.3 Qualitative analysis

The qualitative impact of α and β is expected to be as follows, but further quantitative analysis is needed to check those intuitions and really understand the effect of the parameters on the simulated scenarios.

- **Waiting time** Understood as the time spent by a passenger waiting between the time the request is made and the time it is picked up. It depends only on the pooling parameter, since the rebalancing approach does not allow added delays for the passengers. The higher the ridesharing ($\uparrow \alpha$), the higher the waiting time.
- **Delay time** Understood as the difference of the drop off time and the ideal drop off time, if the passenger had been picked up immediately after the request was made and was driven to its destination with no detours. As the *Waiting time*, it depends only on the pooling parameter, since the rebalancing approach does not allow added delays for the passengers, and the higher the ridesharing ($\uparrow \alpha$), the higher the delay.
- **Vehicle usage** Understood as the actual utilization of the capacity of the car, i.e., the number of passengers a vehicle carries at a certain moment of time, out of the total number of passengers it could carry. The simulations are set with vehicles of capacity 4 (driver non included). As before, it depends only on the pooling parameter, since the rebalancing approach does not consider passenger pooling, and the higher the ridesharing ($\uparrow \alpha$), the higher the vehicle usage.
- **Ignored requests** Depends only on the ridesharing step, but no intuitive conclusion arises.
- **Number of vehicles needed** The fleet size will decrease the more the pooling is encouraged ($\uparrow \alpha$). If vehicles perform several trips instead of assuming a new vehicle carries out each trip, i.e. if relocating and waiting to attend other trips are promoted ($\uparrow \beta$), the total number of vehicles required to attend the same number of requests will decrease.
- **Traffic** Understood as number of vehicles travelling at a given moment of time, or if a certain period of time is being considered, the total time spent travelling by all cars that were active in that interval. While increasing the pooling of passengers ($\uparrow \alpha$) will increase the mean travelling time per vehicle, the overall traffic should decrease. However, increasing the relocation of vehicles ($\uparrow \beta$) will cause more vehicles to travel empty, and thus add traffic.
- **Idling time** Vehicles are only idling while waiting between trips. β determines the maximum travelling time that can be spent by a vehicle relocating between trips. The smaller this limit is, the higher the idling time - both globally and as a fraction of the total time the vehicle is active - will be. Since this limit is proportional to β, a policy that promotes waiting over relocating ($\downarrow \beta$) will increase the total idling time.

  Idling vehicles are assumed to be parked or otherwise out of the street, so the traffic does not increase by their presence. However, while it is assumed that vehicles' engines would be turned off - and therefore, there would be no fuel consumption -, drivers would remain inside their vehicles. This is not included in the cost function because it is assumed that drivers are a "fixed cost" while gas consumption is a variable cost. Moreover, once the vehicle exists (it has been produced), its environmental footprint depends only on the time it is being used.
- **Distance travelled** Directly related to *Traffic*.

5.4 Waiting times and delays

The pooling simulation determines the waiting time tw and delay td for all requests. The upper bound for the waiting time - difference between pick up time and the time at which the request was made - is set to 5 minutes. The maximum delay allowed - difference between drop off time and best possible arrival time - is 10 minutes.

Figure 21 shows the histogram for both waiting time and delay, for $\alpha = \{0.0, 0.2, 0.4, 0.6, 0.8, 1.0\}$. For $\alpha = 1.0$, the pooling is prioritized over the minimization of total delay, so therefore, it is the one with highest total delay, and the maxima is reached. For $\alpha = 0.0$, the waiting time equals the delay, since the only holdback the passenger suffers is the one before being picked up: due to the discretization of the problem, requests made between iterations are considered at the beginning of the subsequent one.

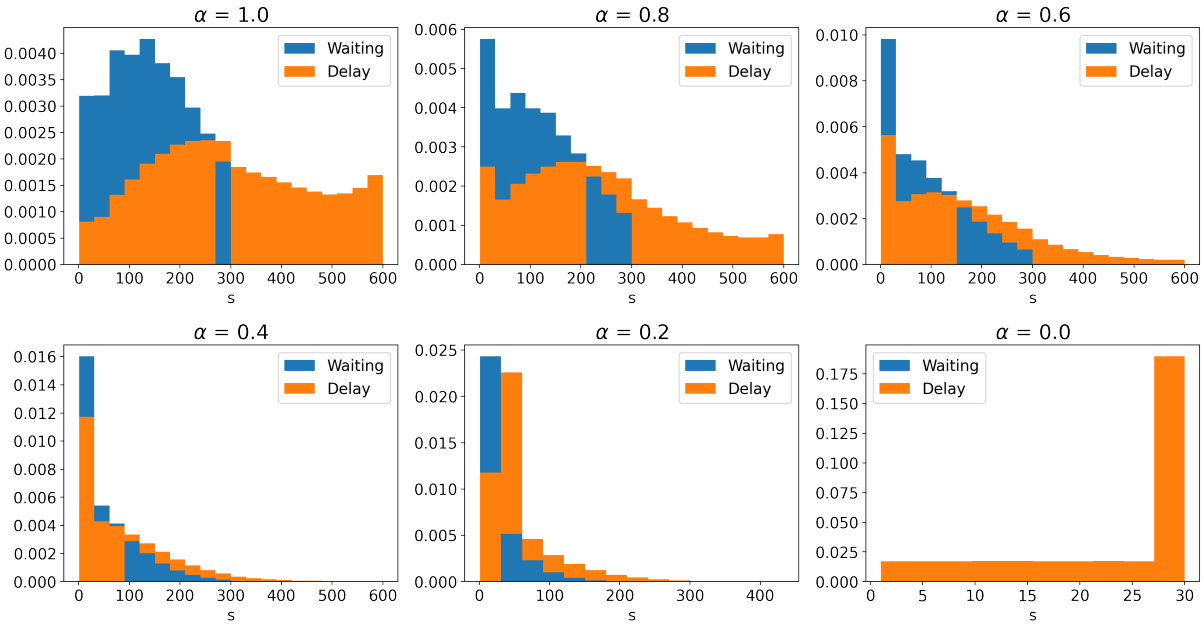

Figure 21: Histograms for waiting times and delays for different α

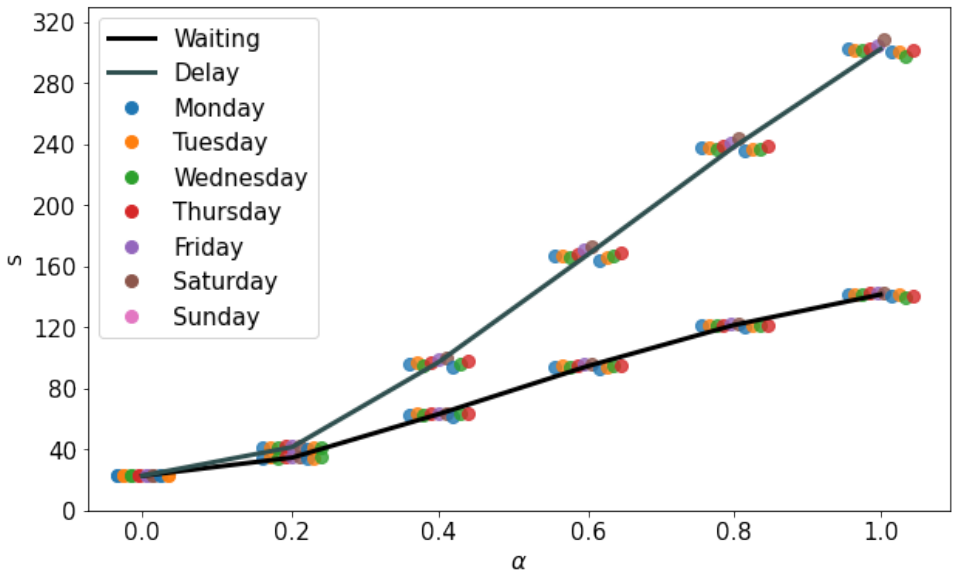

Figure 22: Average waiting times and delays

Figure 22 shows the average for both waiting time and delay. For each α, data points for several days - classified by week day - are plotted. All points centered around a given α correspond to that value of α. They are expanded over a larger range along the x-axis so that the changes between each considered day are appreciable. The average can be performed as the mean for all passengers served during the considered days $(tw(p), td(p))$, or as the means of the averages of the considered days $(tw(m), td(m))$. These quantities are shown in table 9.

α	td(p)	td(m)	tw(p)	tw(m)
1.0	302.04	301.97	141.35	141.32
0.8	238.15	238.05	121.42	121.39
0.6	167.60	167.50	94.57	94.54
0.4	96.97	96.88	63.02	62.98
0.2	41.50	41.46	34.61	34.59
0.0	22.72	22.71	22.71	22.71

Table 9: Average delay per day and per passenger

5.5 Vehicle usage

The pooling simulation determines the usage of each vehicle in terms of seats filled, out of the total capacity. All vehicles considered are assumed to have a capacity $\nu = 4$.

Figure 23 shows the evolution of the mean vehicle usage over the day, computed for each 30 second interval as the weighted average of the number of vehicles with different levels of usage, with the applied weight of the number of passengers being carried. This is equivalent to compute the total number of passengers split between all existing cars.

During the day, between 7am and 11pm, the average vehicle usage remains mostly constant for every α considered. The more requests there are, and the more condensed they are, the easier is to find passengers that can be paired without neglecting the maximum waiting time and delay constrictions. The small spike at the morning rush hour between 7am and 8am, the diminution just after 16h coinciding with the drop on demand (see figure 16) and the generalized smaller usage during night and early morning, are all accounted for by this fact.

Figure 24 shows, for some of the previously considered intervals, the distribution of vehicles per usage and α. All four time stamps are equally distributed along the day, with 6 hour intervals between them.

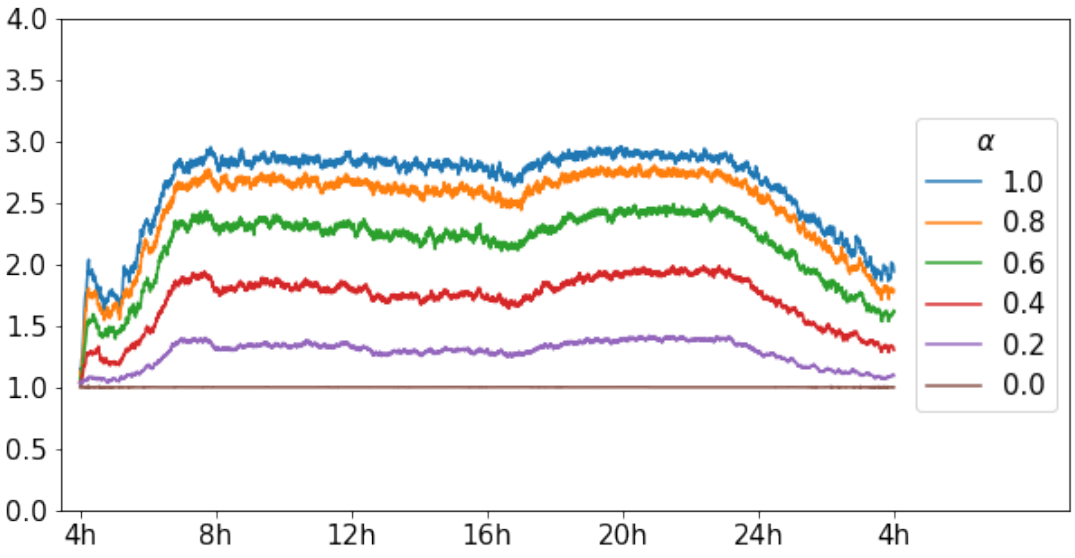

Figure 23: Mean vehicle usage over the day for several α

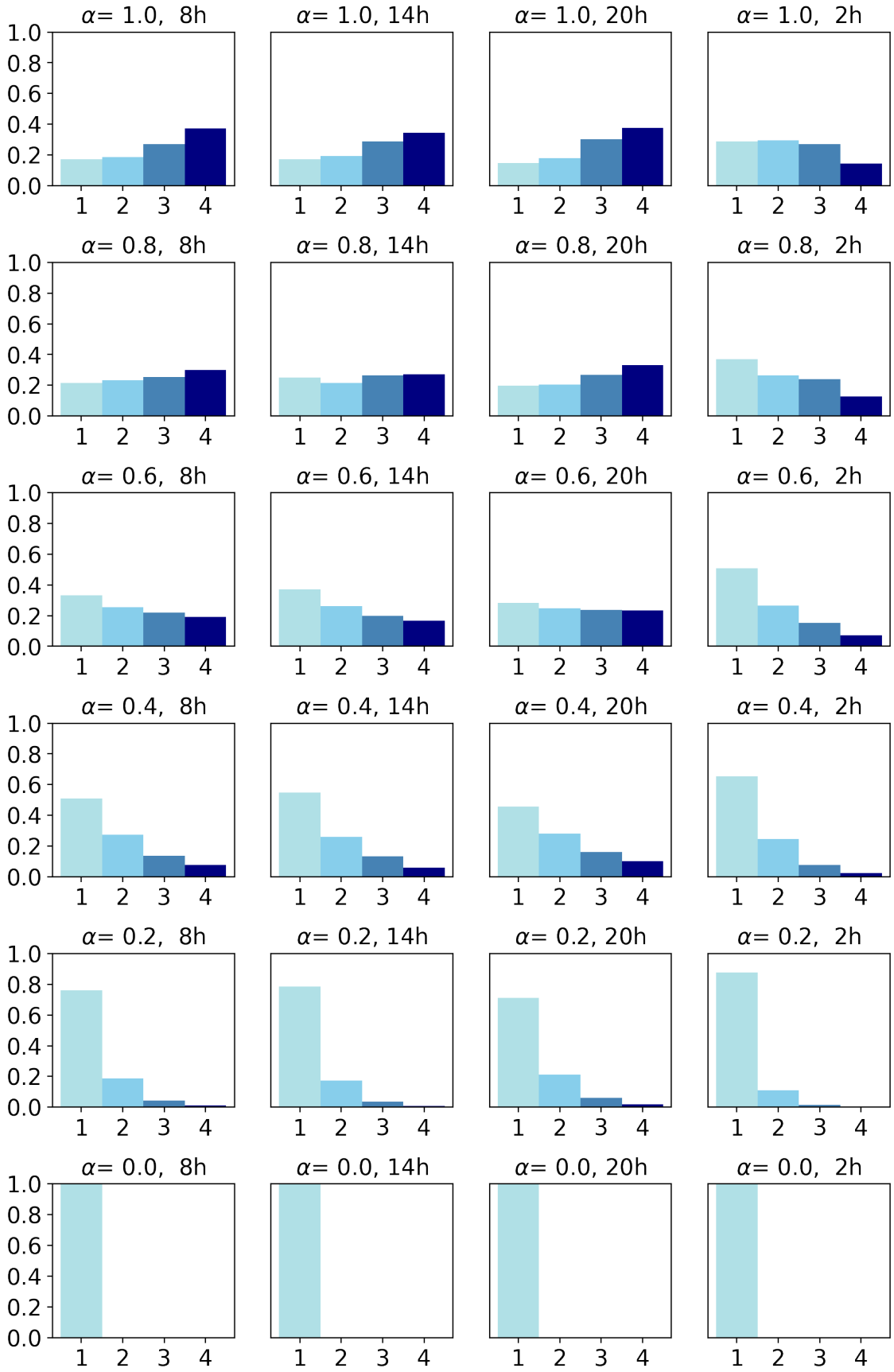

Figure 24: Vehicle usage histograms by time and α

As seen in figure 16, the highest demand point during the day takes place at 8pm; and as seen in figure 23, at that same hour, the highest vehicle usage is reached. This results in 8pm being the set of histograms with a higher relative weight of completely filled vehicles (compared to histograms with same α). To a slightly lesser extent, this also happens for 8am. The opposite case is 2am: its corresponding histograms show the highest relative density of vehicles carrying only one passenger.

Except for $\alpha = 0.0$ - for which the analysis does not apply, since all vehicles carry one and only one passenger - this is true for all values of the pooling parameter α considered.

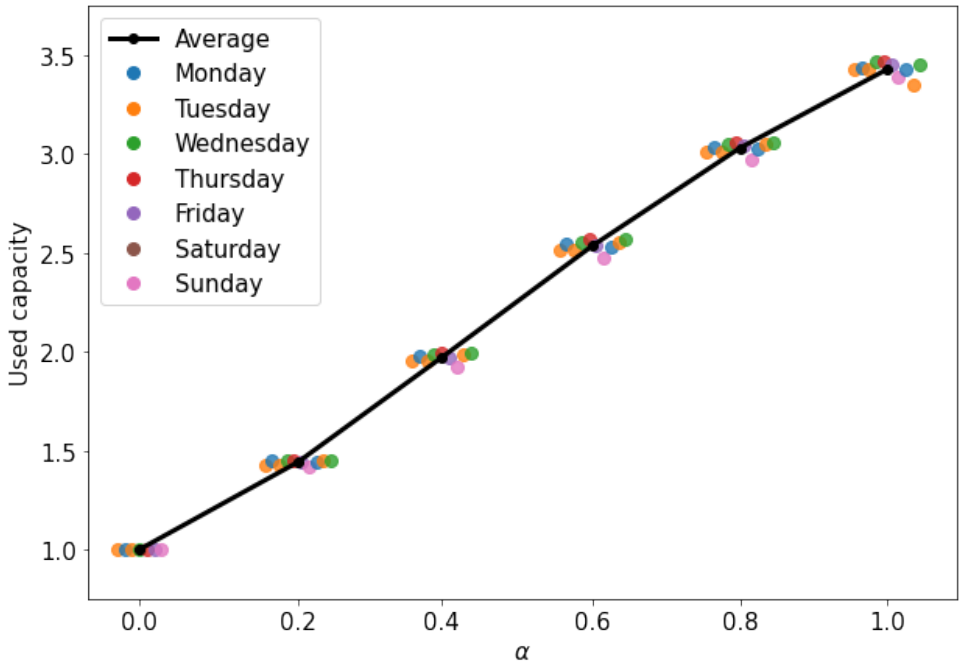

Figure 25: Average maximum usage reached per vehicle

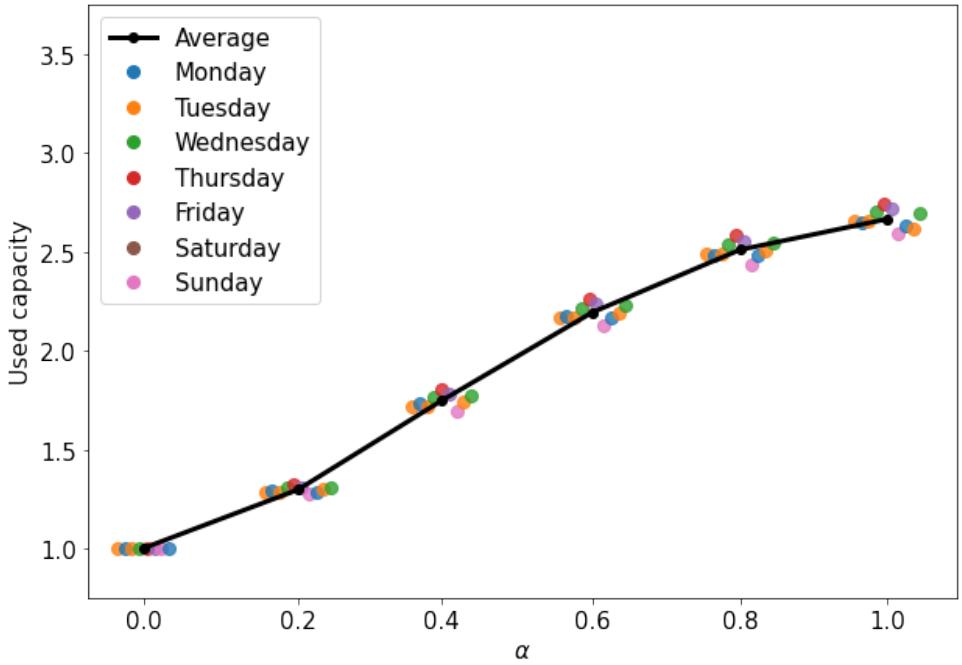

Figure 26: Average mean usage during the day

Figure 25 shows the average over all vehicles of the maximum usage reached by them. At $\alpha = 1.0$, an average of maximum usage of approximately 3.5 means that most vehicles end up carrying 3 or 4 passengers at once, at some point of their existence. This quantity grows almost linearly with α.

Figure 26 shows the average over the day of the mean usage. At $\alpha = 1.0$, an average of mean usage of approximately 2.7 means that, even though most vehicles end up carrying at least 3 passengers at once, this is not sustained during all the existence of the vehicle. Therefore, this quantity grows much slower with α, compared to the one shown in figure 25.

For both figures, all points centered around a given α correspond to that value of α, and are expanded over a larger range along the x-axis so that the changes between each considered day are appreciable.

The cost function seen in equation 10 shows that α determines which of two opposing goals is prioritized: decreasing the total delay versus decreasing the traffic. Figure 27 and figure 28 show the effect of setting α to certain values on these parameters:

More pooling $\leftrightarrow$ Higher occupancy per vehicle $\leftrightarrow$ More delay per passenger $\leftrightarrow$ Less total traffic

Figure 27 shows, in absolute numbers, the relation between the mean occupancy of vehicles over the day and the mean delay per passenger. As seen in the previous relation, these two quantities are directly related, so it is expected that, as one grows, so does the other.

Figure 28 shows the relation between the percentage increase of travelling time per passenger and the percentage negative increase (i.e. decrease) of the of the total time spent travelling by all vehicles, which summarizes the global traffic. These two quantities are inversely related.

The baseline for both quantities is set by assuming that each vehicle carries only one passenger, and that there are enough vehicles available so that requests can be immediately picked up and dropped of without waiting times or delays.

For $\alpha = 0.0$, as seen on figures 23-28, the mean usage is 1, since only requests with the same initial and final stations, and ordered in the same 30 second interval could be pooled. Intuitively, this should lead to no delay for all passengers, but figure 22 and figure 27 prove otherwise. This is due to the fact that the implementation considers a discretization of time in intervals of duration 30 seconds. Trips can only start at time stamps $30, 60, 90, \cdots$, considering respectively requests announced within the intervals: $[0, 30), [30, 60), [60, 90), \cdots$ Therefore, since all passengers are the first (and only) passenger in their trip, and no trip can start exactly at the time its first request was made, all passengers have some associated delay to them, to an average of just over 20 seconds per passenger.

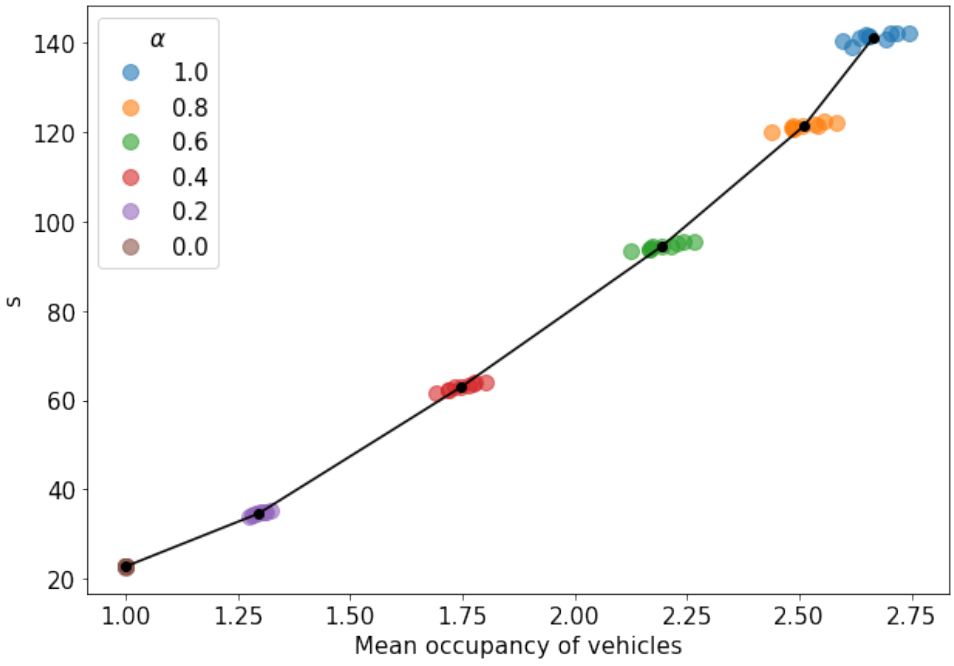

Figure 27: Pareto curve: mean occupancy vs. delay

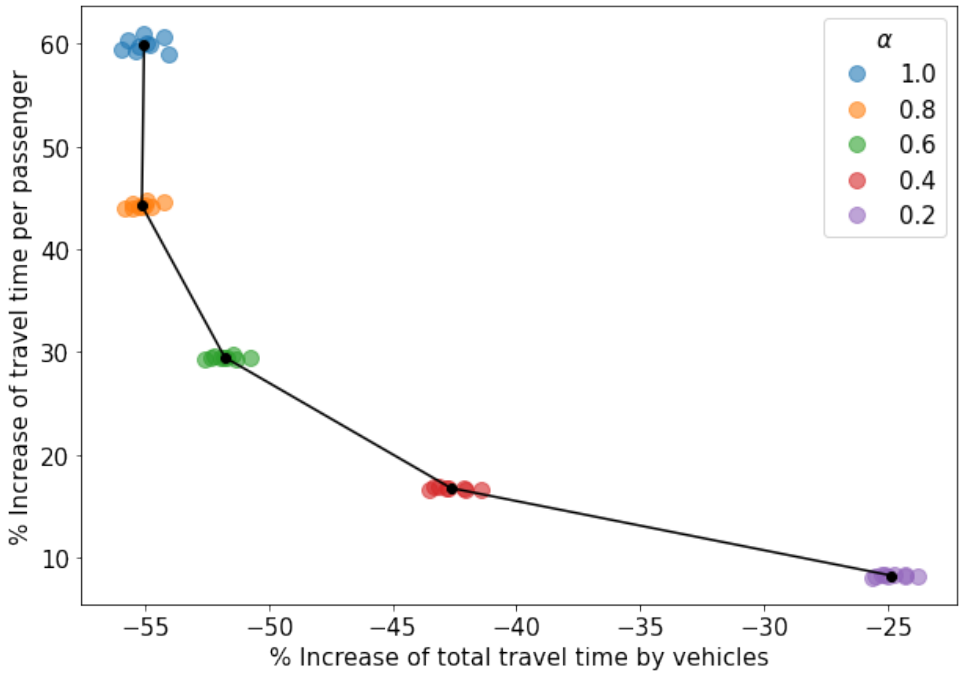

Figure 28: Pareto curve: increase in travel time vs increase in delay

5.6 Ignored requests

Less than 0.1% requests are rejected. Figures 29-30 show that a larger α seems to mean more rejections, but the day being considered constitutes a greater influence.

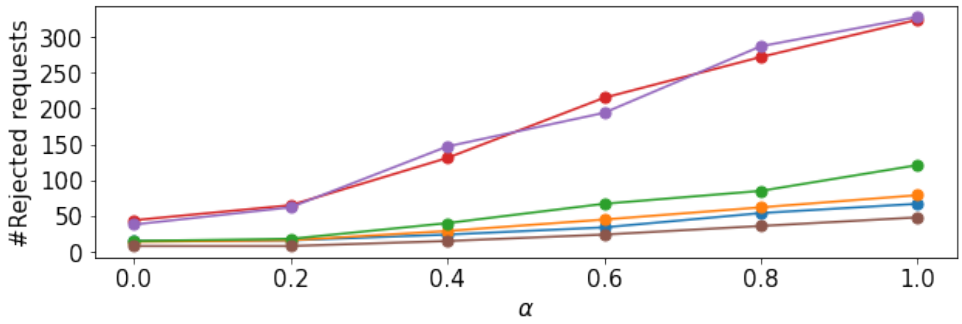

Figure 29: Total rejected requests per simulation and α for 6 days

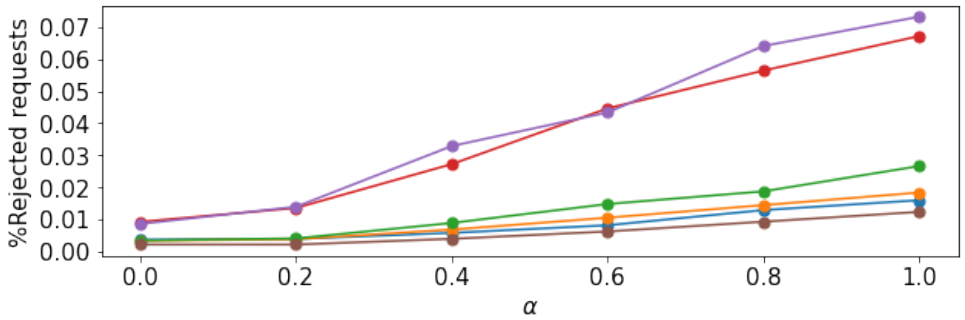

Figure 30: Percentage of rejected requests per simulation and α for 6 days

5.7 Number of vehicles

The pooling code assigns several passengers to the same trip, and then, the rebalancing code assigns several trips to a single vehicle. Table 10 shows how this affects to the number of vehicles needed after each part of the code is run, for the 19th week of the year. The baseline (base) is assumed to be the number of accepted requests for a given day and pooling parameter - one passenger per trip, one trip per vehicle. The fleet size required decreases to the next columns when the ridesharing (rs) and rebalancing (β) are run. The evolution of how the number of vehicles decreases with this simulations for a given day is shown in figure 31. While setting the pooling parameter to $\alpha = 1$ makes a big impact in the initial number of vehicles needed (central point of first subfigure), running the rebalancing decreases significantly the range of fleet size (for $\beta = 0$, right point of first subfigure and leftmost point of second subfigure), to the point where, if rebalancing is run for $\beta = 1$, the effect of α gets reduced to the order of magnitude of thousands, instead of hundred of thousands when only considering the pooling, shown in figure 32 (see also figures 33, 36, 36). The number of vehicles decreases exponentially with α (fitting: $y = e^{x*-1.78+12.96}$, grey line).

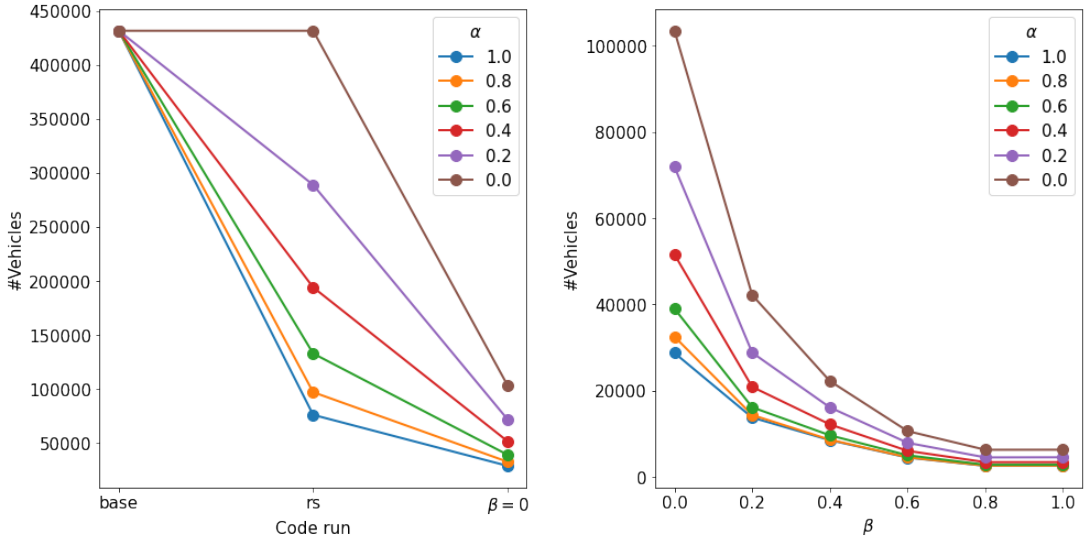

Figure 31: Number vehicles needed: baseline, ridesharing and rebalancing

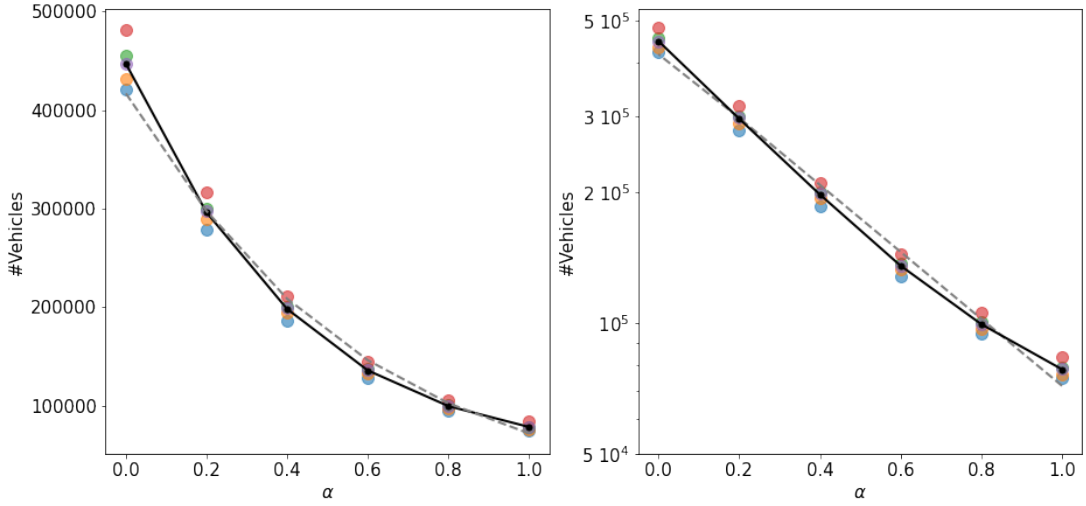

Figure 32: Pooling and number vehicles needed

Week_Day_α	base	rs	$\beta = 0.0$	$\beta = 0.2$	$\beta = 0.4$	$\beta = 0.6$	$\beta = 0.8$	$\beta = 1.0$
19_0_1.0	391736	70070	27376	13311	8503	4411	2391	2390
19_0_0.8	391748	88598	30318	13851	8726	4439	2377	2381
19_0_0.6	391760	120901	37117	15973	9884	4787	2619	2633
19_0_0.4	391769	175469	48278	20434	12164	5879	3193	3213
19_0_0.2	391776	261230	27376	13313	8498	4411	2391	2392
19_0_0.0	391776	391494	95037	38199	21656	9655	5976	5908
19_1_1.0	421246	74334	28733	14042	8734	4535	2491	2500
19_1_0.8	421259	94096	31961	14481	8909	4605	2524	2560
19_1_0.6	421279	127980	38720	16532	9887	4965	2755	2784
19_1_0.4	421289	185928	50440	20888	12372	6004	3323	3378
19_1_0.2	421297	277717	70201	29267	16559	8111	4448	4530
19_1_0.0	421298	420874	102755	41225	21994	10842	6290	6267
19_2_1.0	431503	75656	28742	13753	8499	4475	2552	2552
19_2_0.8	431520	96660	32522	14348	8605	4523	2584	2595
19_2_0.6	431537	132458	39024	16135	9628	5000	2793	2810
19_2_0.4	431553	193662	51586	20866	12161	6019	3399	3406
19_2_0.2	431566	288479	71929	28812	16105	7886	4499	4531
19_2_0.0	431568	431132	103502	42182	22219	10606	6273	6288
19_3_1.0	455062	78357	29941	14577	8904	4812	2638	2644
19_3_0.8	455098	100263	33750	14922	8945	4864	2638	2652
19_3_0.6	455116	136909	40509	16863	10007	5281	2862	2881
19_3_0.4	455143	199649	53247	21579	12489	6597	3454	3488
19_3_0.2	455165	299666	74983	29973	16477	8884	4608	4728
19_3_0.0	455168	454419	110270	45002	22995	11834	6483	6561
19_4_1.0	481433	82687	31271	14704	8320	5069	2625	2629
19_4_0.8	481485	104985	34759	15048	8404	5138	2649	2660
19_4_0.6	481542	143117	41767	16983	9323	5601	2856	2881
19_4_0.4	481626	209813	55037	21592	11717	7026	3466	3479
19_4_0.2	481692	314959	78075	30762	15800	9681	4633	4650
19_4_0.0	481713	481023	114603	44709	21764	13013	6422	6482
19_5_1.0	447103	77308	28901	13117	8096	4905	2488	2511
19_5_0.8	447144	98321	31699	13604	8175	5002	2523	2519
19_5_0.6	447237	134721	38773	15267	8903	5444	2718	2730
19_5_0.4	447284	197610	50634	19055	11141	6736	3292	3314
19_5_0.2	447369	295700	70322	25314	14825	9065	4444	4451
19_5_0.0	447393	445251	101057	35603	20427	12815	6288	6296
19_6_1.0	307422	57907	22941	10311	6541	3821	1938	1940
19_6_0.8	307431	74948	26231	10921	6699	3813	1994	1996
19_6_0.6	307438	102593	32036	12530	7438	4119	2190	2199
19_6_0.4	307450	148018	41654	15744	9291	5086	2635	2659
19_6_0.2	307457	214580	56054	20508	11738	6419	3456	3506
19_6_0.0	307461	307153	76163	27729	15830	8551	4721	4784

Table 10: Number of vehicles needed. One-week simulation

Averaging over all days considered for all pairs of values of α and β, the results shown in figure 33 are obtained. For a fixed value of α, the dependency of the fleet size required with β is also exponential (with a smaller - in absolute value - rate of growth than the previous case). Figure 34 shows the goodness of fit (ignoring $\beta = 1$ in grey, and $\beta = 0,1$ in black).

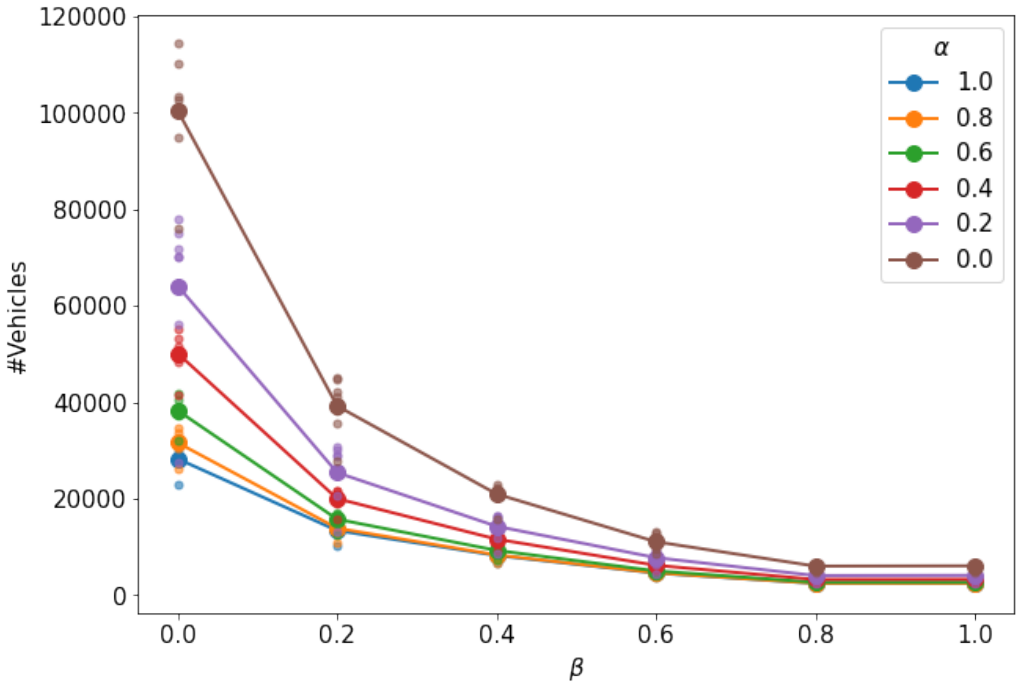

Figure 33: Fleet size needed per α, β

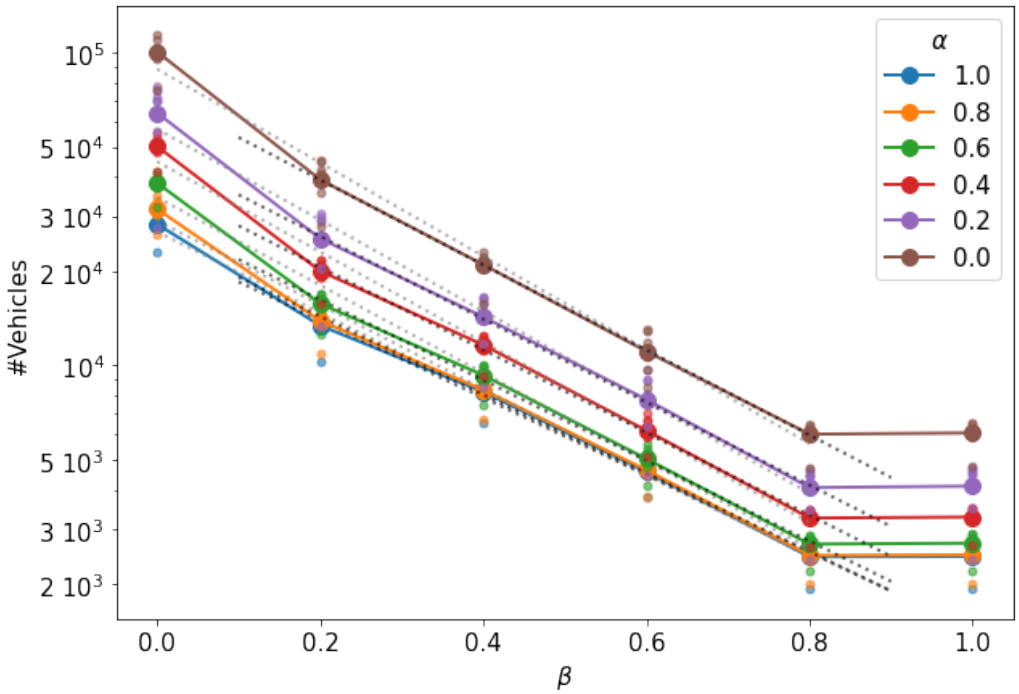

Figure 34: Fleet size needed per α, β. Log-scale.

The density of requests varies during the day. Since the number of vehicles at any given moment of time is directly related to the number of requests, the number of vehicles during the day fluctuates. Figure 35 shows the evolution of the number of vehicles as a result of the pooling of passengers, while figure 36 shows the results of the rebalancing of vehicles.

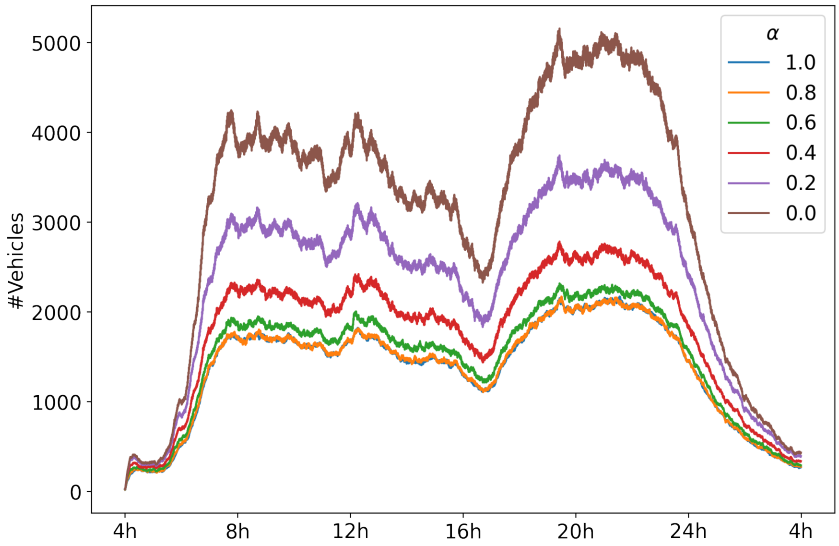

Figure 35: Evolution of the number of vehicles during the day after pooling

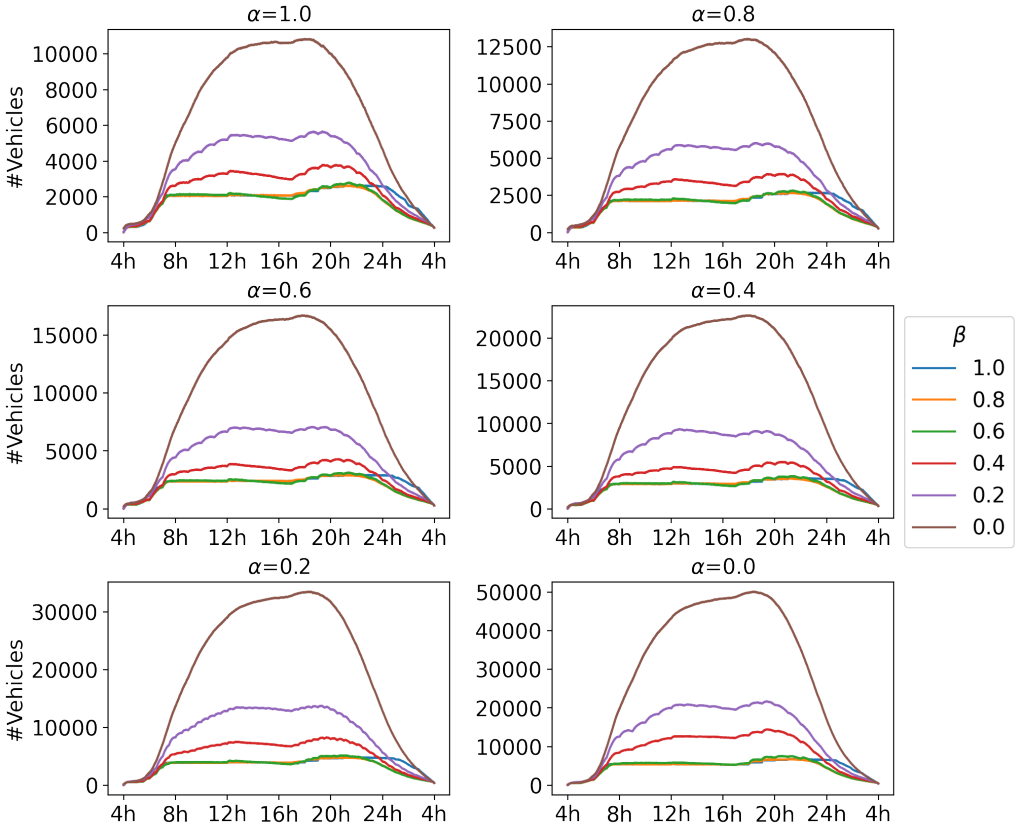

Figure 36: Evolution of the number of vehicles during the day after rebalancing

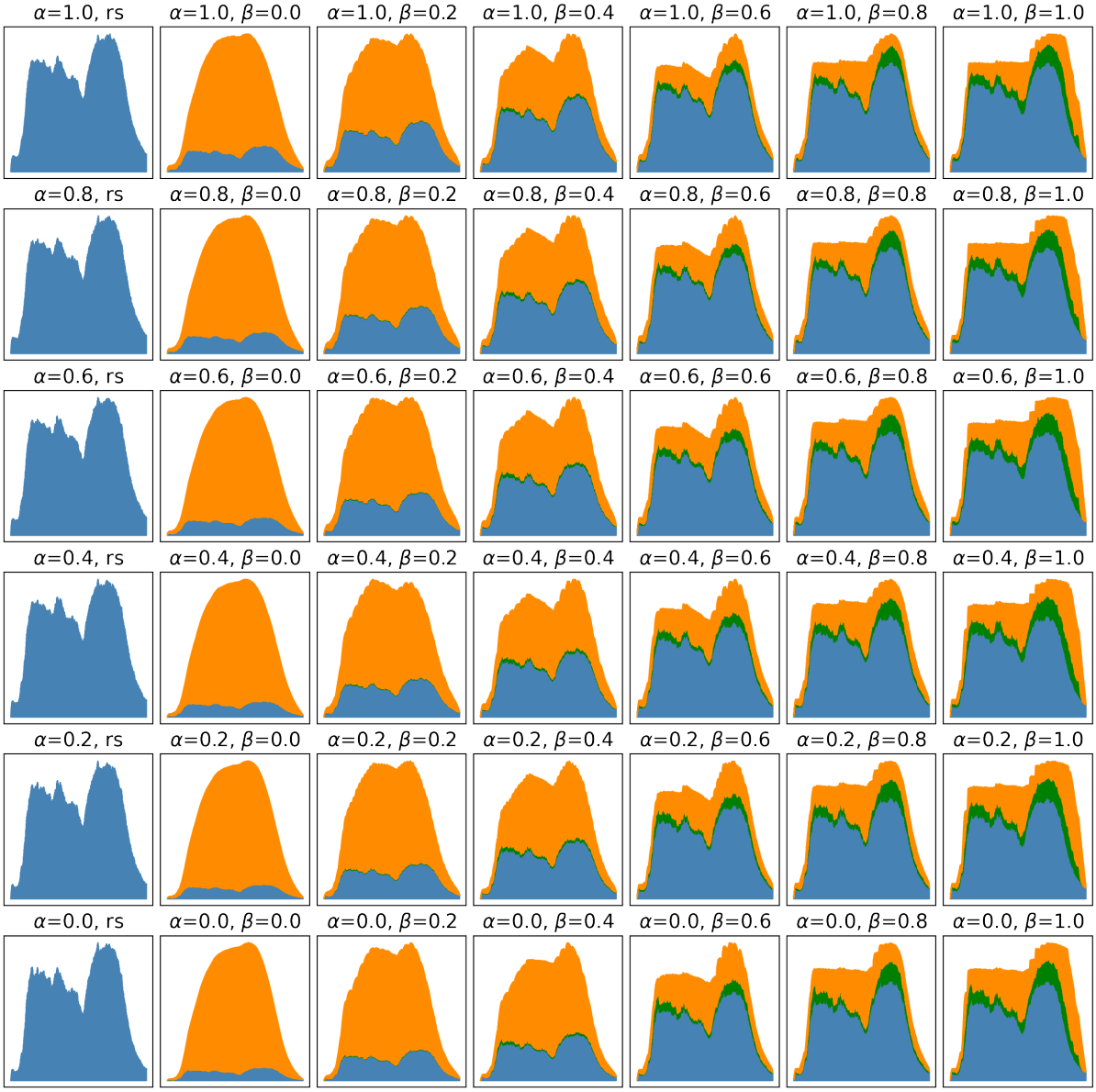

Figure 37: Proportion of vehicles carrying passengers (blue), relocating (green) or idling (orange)

For a fixed α, since all feasible connections for β_1 are also feasible $\forall\beta_1$ with $\beta_1 \geq \beta_0$, larger β yield smaller fleets. The overall shape - not taking into account the absolute value - of the curve of vehicles needed during the day for a given β remains mostly the same for all values of α. As does the proportion between fleet sizes between different values of β. Therefore, it can be sustained that the rebalancing code has a "correlative" effect on all set of trips, and that α mainly determines the dimension of the fleet.

Figure 37 shows the how the studied fleet can be sorted depending on the task each vehicle is performing: vehicles carrying passengers are shown in blue, vehicles relocating between trips are displayed in green, and vehicles idling, waiting for their next assigned trip to begin, are shown laid out in orange. The figure only shows the proportion between each class within a (α, β) pair, but not the relative numbers of total vehicles between those parameters (that can be found in figures 35-36). Rows show results for a fixed value of α, and the distribution of passengers remains constant within it. The fact that all rows have very similar distributions supports the past inference that rebalancing has an akin effect on a set of trips, regardless of α.

5.8 Traffic and idling time

Traffic is measured as the amount of time the set of vehicles spends travelling. Figure 38 shows the average of total traffic that results from running the pooling code for several days of data, for the 6 established values of α. This is taken as a baseline for the next two figures, which also take into account rebalancing.

A vehicle rebalancing may perform two types of actions: relocating - going from one station to another, where it will begin another trip - or idling - once it has arrived to the goal station, it has to wait to pick up the first passenger. Only the relocation portion of the rebalancing rises the traffic. The increment of traffic due to the relocation of vehicles is shown in figure 39.

If vehicles idling are also considered, then the rise of presence of vehicles is shown in figure 40. The increase range is greater for this case, but rapidly decreases with β. As seen on figure 37, most vehicles are idling fore small values of β, and there are always more vehicles idling than relocating. This is due to the fact that the rebalancing cost function balances two opposing goals: decreasing the number of vehicles needed, while avoiding a rise in traffic. Moreover, the cost function sets an implicit maximum relocating time allowed. Idling does not increase the traffic, and avoids needing extra vehicles, so it is encouraged by the cost function. In the extreme case $\beta = 0$, all vehicles rebalancing are idling: they wait in their last station for some other trip to start from it.

Figure 42 shows the relation between total time spent relocating and idling for all rebalancing vehicles, for α, β pairs. Table 11 shows the average relocating time per vehicle, average idling time per vehicle, maximum relocating time for all vehicles considered, and maximum idling time. The maximum relocating time allowed is given by figure 41.

Figure 43 shows the relation between the number of vehicles needed and the increase in traffic (measured as travelling vehicles). The rate of time idling and relocating is the same for all values of α, for small values of β. For values of β larger than 0.6, the idling time remains mostly constant, while the relocating time increases. Moreover, for $\beta = 0.8$ and $\beta = 1.0$, the required fleet size is virtually the same, while the increase in traffic is around a 5% more for $\beta = 1.0$. In fact, for some of the simulations ran, the number of vehicles needed for $\beta = 0.8$ is smaller than the one required for $\beta = 1.0$ (example: table 10, week 19, day 0, $\alpha \in \{0.2, 0.4, 0.6, 0.8\}$). While the difference between both results is almost negligible, it seems counterintuitive. This is difference is due to the fact that the rebalancing code is not run for the entire set of trips, but for smaller intervals in time, whose results are later merged.

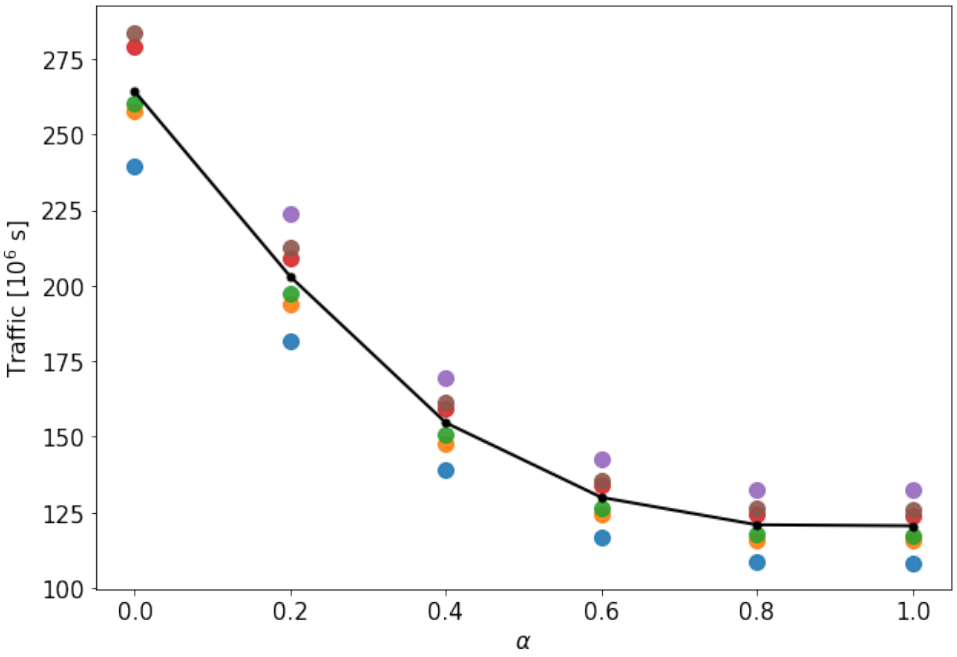

Figure 38: Pooling and traffic

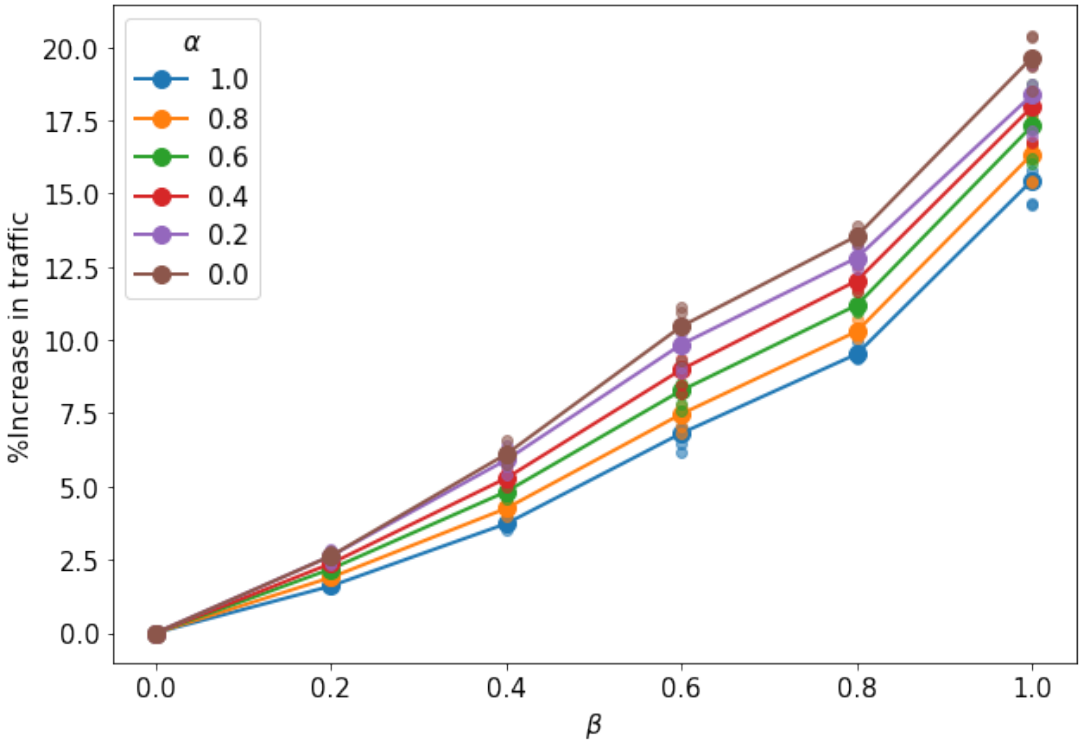

Figure 39: Increment in traffic due to relocation of vehicles, per α, β

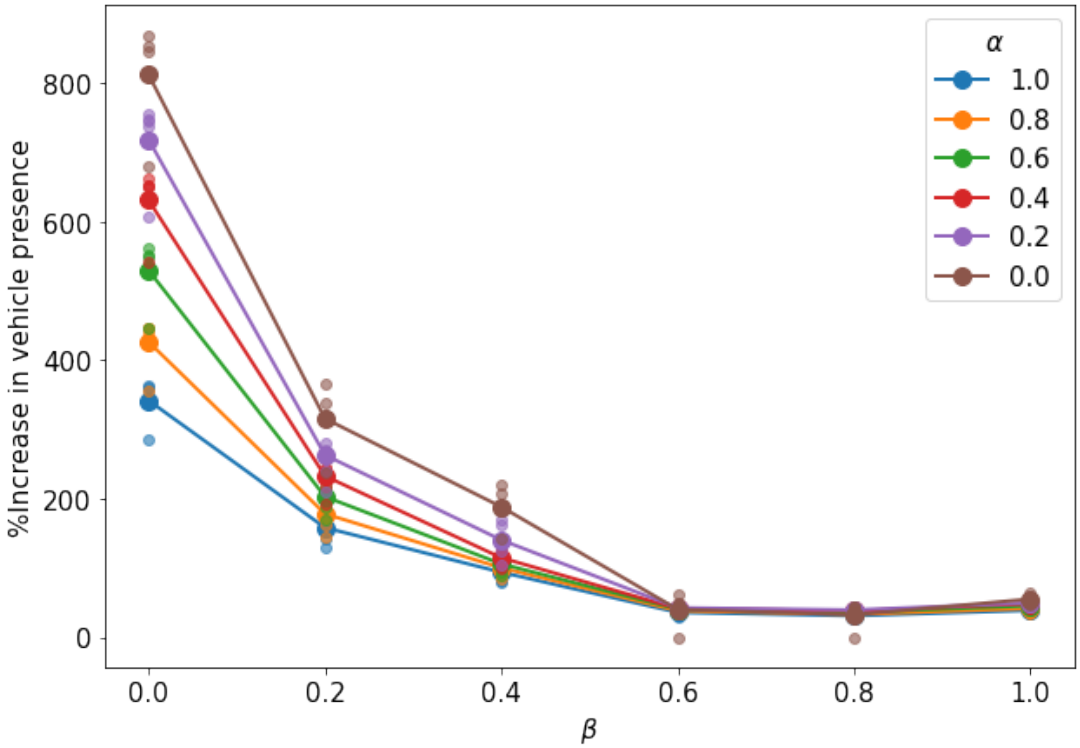

Figure 40: Increment in presence of vehicles in the streets due to rebalancing, per α, β

		β					
		0.0	0.2	0.4	0.6	0.8	1.0
α	1.0	0.00	30.93	67.19	115.63	150.39	247.17
	0.8	0.00	27.86	59.24	98.99	127.42	207.67
	0.6	0.00	24.44	51.49	84.62	108.42	174.14
	0.4	0.00	21.27	45.63	75.14	94.91	146.78
	0.2	0.00	21.73	46.33	74.44	87.45	130.20
	0.0	0.00	16.93	38.70	67.81	82.09	118.22

		β					
		0.0	0.2	0.4	0.6	0.8	1.0
α	1.0	8866.13	3208.97	1359.19	373.58	308.76	364.74
	0.8	8144.19	2695.88	1141.50	332.27	267.48	321.66
	0.6	7470.76	2348.27	873.13	280.11	232.75	287.81
	0.4	6908.76	2122.42	797.76	243.16	199.52	251.67
	0.2	6785.83	2098.23	854.41	207.09	176.83	228.29
	0.0	6719.55	2250.63	1127.07	238.64	172.68	221.22

		β					
		0.0	0.2	0.4	0.6	0.8	1.0
α	1.0	0	150	399	897	1948	2621
	0.8	0	150	400	897	2028	2537
	0.6	0	150	400	900	1899	2497
	0.4	0	150	399	896	1914	2451
	0.2	0	150	400	898	2025	2557
	0.0	0	150	400	897	2234	2616

		β					
		0.0	0.2	0.4	0.6	0.8	1.0
α	1.0	86370	75638	70628	62854	30171	3600
	0.8	86310	75025	71273	52745	31263	3600
	0.6	86370	74970	70733	27774	29837	10193
	0.4	86370	76256	71277	32938	37602	10158
	0.2	86370	73925	72968	62391	36546	3600
	0.0	86340	75924	70460	52190	41195	3600

Table 11: (1) Average relocating time per vehicle, (2) average idling time per vehicle, (3) maximum relocating time, (4) maximum idling time

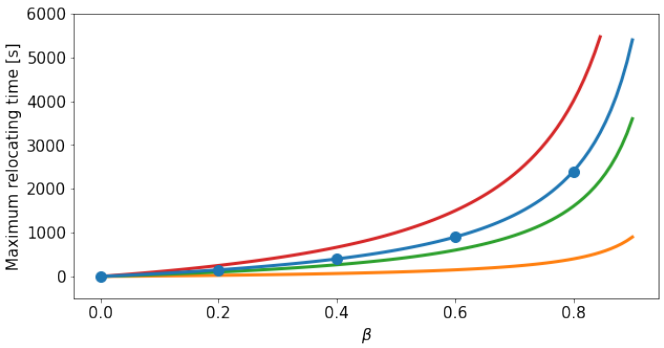

Figure 41: Maximum relocating time per β, for $\delta = [100, 400, \mathbf{600}, 1000]$

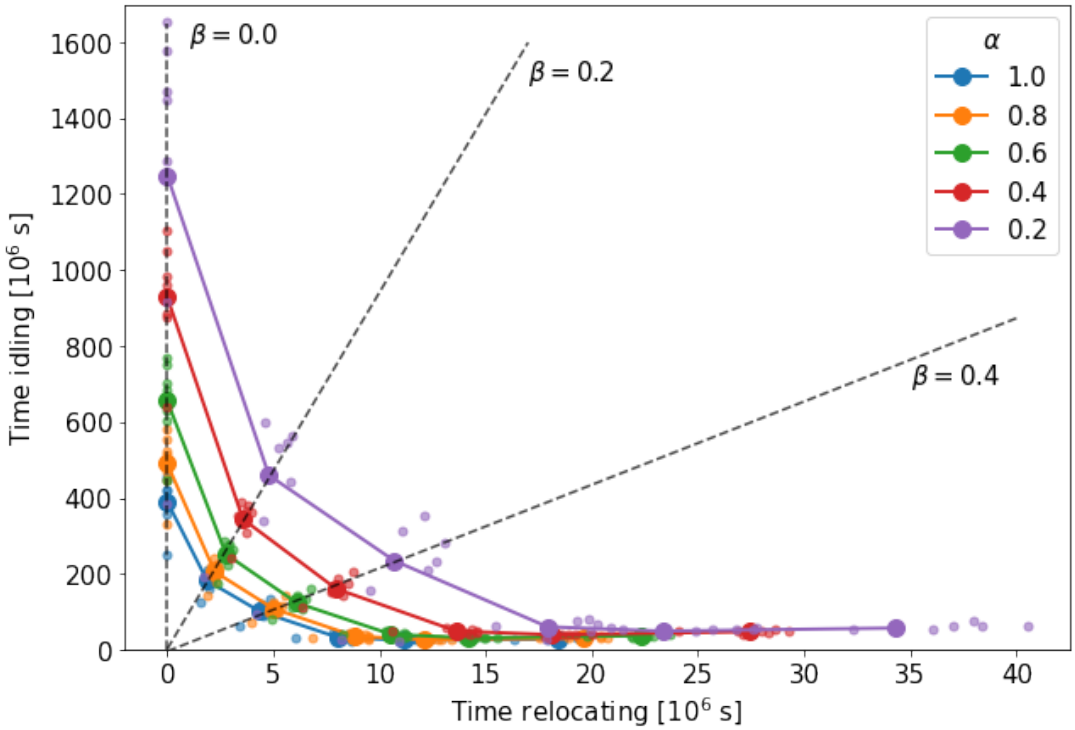

Figure 42: Relation between time spent relocating and idling

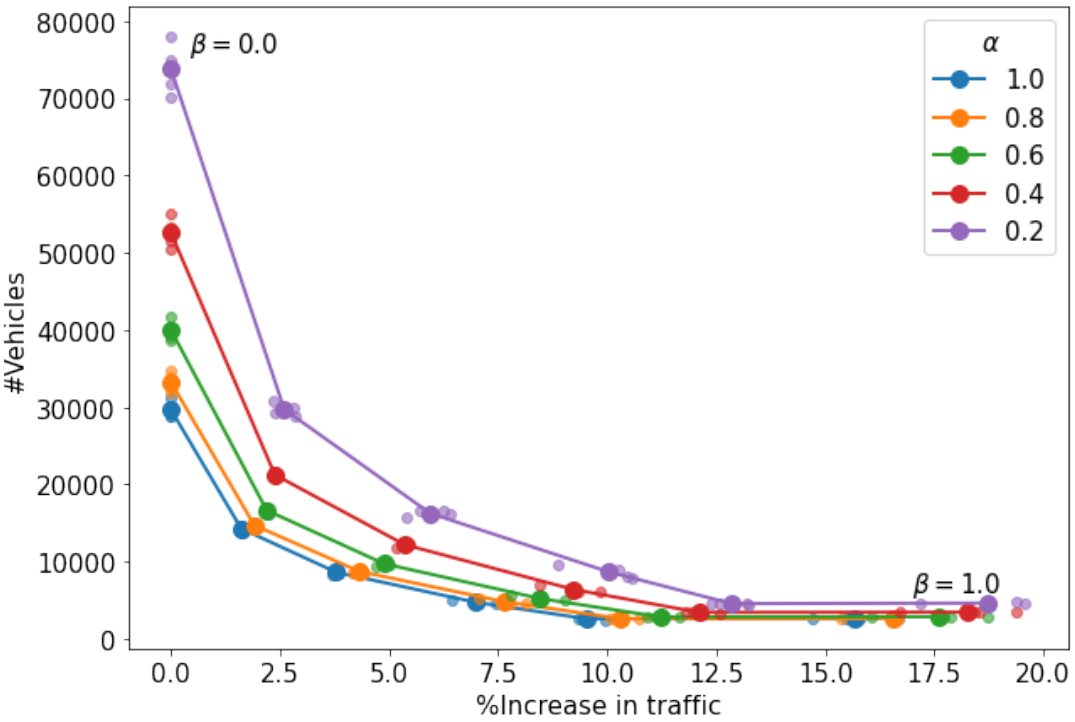

Figure 43: Pareto curve: increase in traffic vs number of vehicles

The traffic is directly related to the number of vehicles active at any given moment of time. Therefore the previously seen evolution over a day hints to how traffic evolves during the day. If traffic is understood only as vehicles moving - and ignoring idling vehicles - then figure 37 can be interpreted as the proportion between empty and non-empty vehicles. By deleting those idling vehicles from figure 36, figure 44 is obtained. As before, all profiles look the same except for the scale.

Figures alike 33 and 34 can be produced if, instead of plotting the total number of vehicles needed, the maximum number of vehicles needed at once is graphed. These results can be found in figures 45 and 46. While the exponential relation between this parameter and β is not as noteworthy as with previous cases, the evolution of this curve between β seem to run parallel.

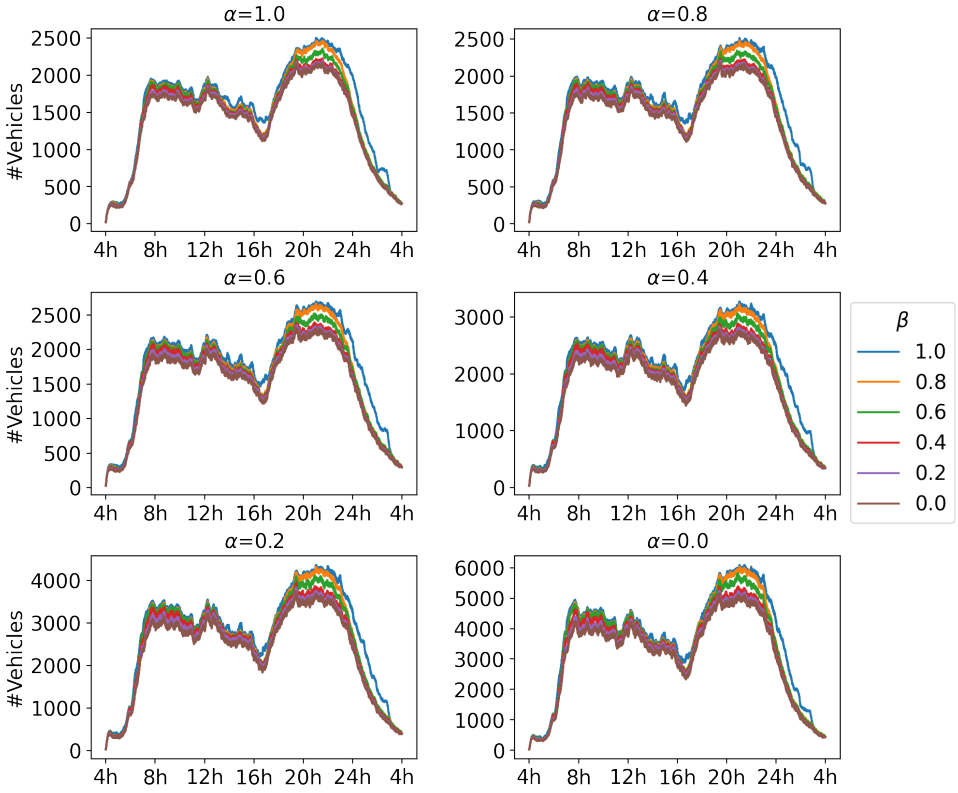

Figure 44: Evolution of the number of vehicles during the day after rebalancing

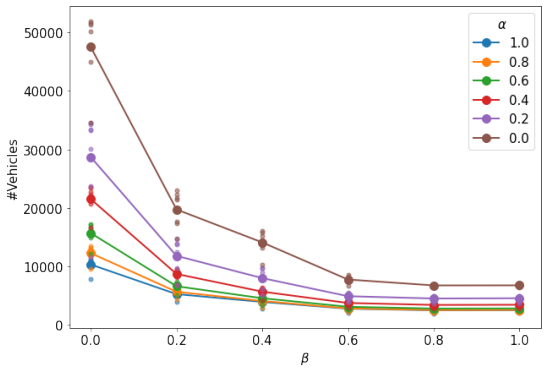

Figure 45: Maximum number of vehicles at once per α, β

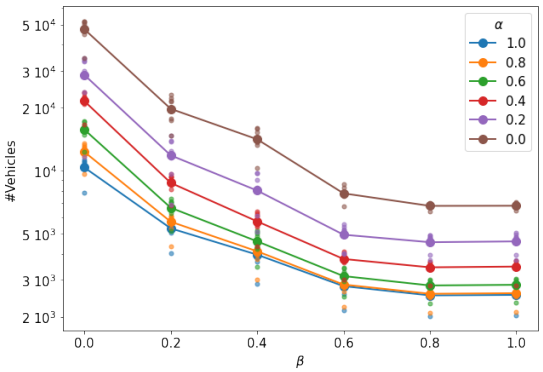

Figure 46: Maximum number of vehicles at once per α, β. Log-scale.

5.9 Rebalancing

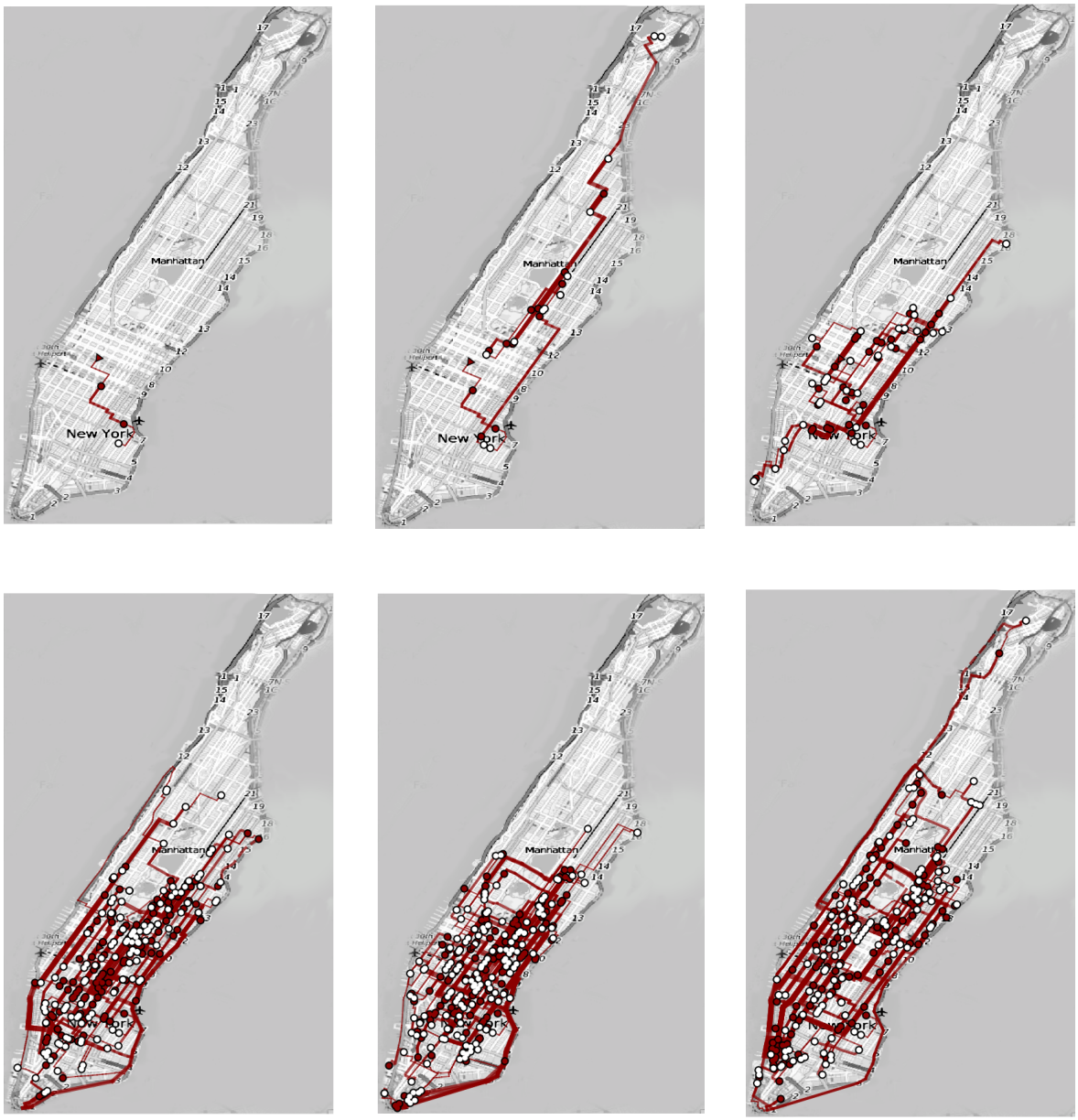

Figure 47: Path of a single vehicle, for different values of β (top: 0.0, 0.2, 0.4; bottom: 0.6, 0.8, 1.0)

The effect of the rebalancing on different statistics has already been analyzed on previous sections. However, it is interesting to see what is the effect of applying one or another β to the behaviour or a vehicle.

Figure 47 focus on an existing vehicle at a moment of the day, and explores the paths of all the future trips it will perform. For $\beta = 0$, a single trip with two passengers is carried out; but when β increases, the number of chainings between trips grows, to the point where the path becomes indecipherable.

As with figure 49, the width of the line is proportional to the number of passengers it will carry out in that interval, and the dash line represents the vehicle rebalancing. The filled dots represent pick-up points, while the empty ones depict drop-off points. The triangle represents the current position of the vehicle, with its tip pointing to the direction of motion. Figure 48 shows the overall state of the fleet at the same time stamp of figure 49, at around 2am.

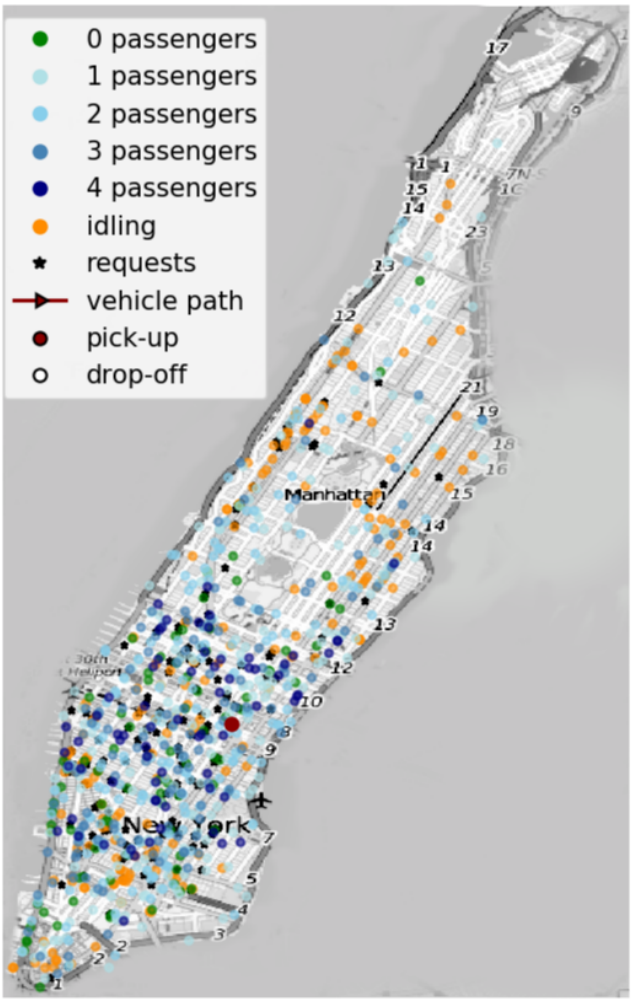

Figure 48: Geographical distribution of vehicles per capacity usage and rebalancing action

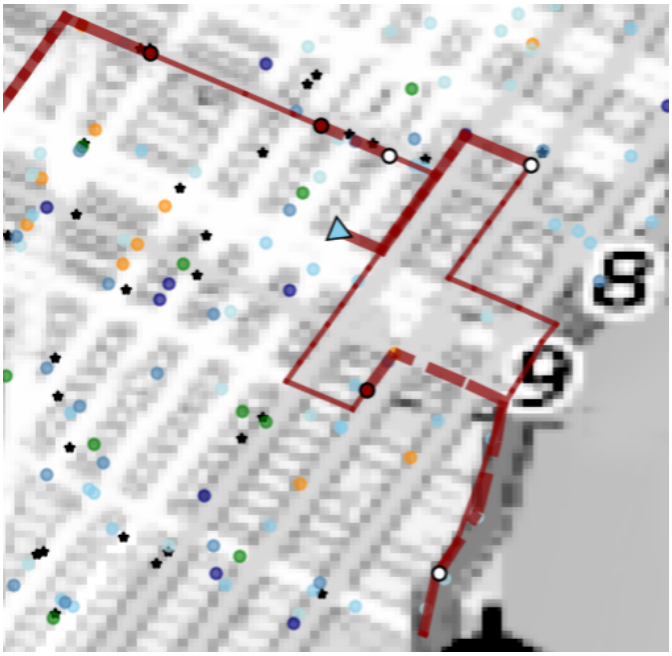

Figure 49: Path of a give vehicle

6 Conclusions and next steps

The simulations show that the almost half a million requests made daily in Manhattan, could be performed by a fleet of less than 3000 vehicles, with a maximum waiting time of 5 minutes, and delay of 10 minutes. If the goal is to decrease the number of vehicles needed, then $\alpha = 1.0$ would be chosen, and since $\beta = 0.8$ has comparable outputs to $\beta = 1.0$ in terms of fleet size, and a smaller total traffic, the final selection would be $\alpha = 1, \beta = 0.8$.

While this approach yields useful results for the estimation of the fleet size required to serve the daily set of requests at Manhattan, the rebalancing code is not dynamic: it uses the global knowledge of the parameters of the trips at the end of the day to match them to vehicles. The dynamic approach was one of the main characteristics of the ridesharing part, and it is the one that describes an actual implementable solution.

Technically, $\beta = 0.0$ is dynamic, since it instructs all vehicles in its fleet to wait where they finished until the next trip starts from that station. However, this leads to average idling times higher than an hour and a half, with extreme cases in which the waiting time for the vehicle exceeds 20 hours. And these massive idling times are not only present for the case $\beta = 0.0$. For $\beta = 0.2$, the average idling time rounds the 40 minutes, and for $\beta = 0.4$, it almost reaches the half hour. While the average for $\beta = 1.0$ does not stretch to 5 minutes, there are vehicles with idling times longer than two and a half hours.

This is due to the fact that the idling time is not considered in the rebalancing cost function, and that while the firsts assignments are made in smaller time windows, there is a second global iteration that considers all unassigned trips. The impact is more significant on simulations with lower β, in which the relocating time allowed is small, but it may affect larger β if not all trips are combined within the first iteration.

There are a couple of approaches to solve this. The first one could be including the idling time in the rebalancing problem statement, either as a part of the cost function, or as a part of the restrictions applied to consider that a matching is feasible.

The first system would allow to assign different weightings to idling and relocating times, and it would set an implicit maximum combination of them. For example, if the cost function was changed from $\sum_F (1-\beta) \cdot c_{ij} - \delta \cdot \beta$, where F represents the set of feasible edges (i, j) and c_{ij} is the travelling time associated to said edge, to

$$\sum_F (1-\beta) \cdot (\gamma_1 \cdot c_{ij} + \gamma_2 \cdot c'_{ij}) - \delta \cdot \beta \tag{34}$$

where c'_{ij} represents the idling time of a vehicle performing trip j just after having carried out trip i. Like with the original cost function, since the sum is being minimized and there is always the option of not assigning two trips to the same vehicle with cost zero, only negative summands will be apt to be chosen:

$$(1-\beta) \cdot (\gamma_1 \cdot c_{ij} + \gamma_2 \cdot c'_{ij}) - \delta \cdot \beta \leq 0 \Rightarrow \gamma_1 \cdot c_{ij} + \gamma_2 \cdot c'_{ij} \leq \frac{\delta \cdot \beta}{1-\beta} \tag{35}$$

If the values of δ and β are fixed, then the feasible combinations of c_{ij} and c'_{ij} are points in the plane $c_{ij} - c'_{ij}$ belonging to the surface encompassed by the lines: $c_{ij} = 0,\ c'_{ij} = 0,\ c'_{ij} = \frac{1}{\gamma_2}\left(\frac{\delta \cdot \beta}{1-\beta} - \gamma_1 \cdot c_{ij}\right)$. The original implementation set $\gamma_1 = 1, \gamma_2 = 0$. With both cost functions, the added cost of assigning an edge with cost zero and not assigning it are the same. This can be solved by giving zero-cost edges an artificial value of $-\epsilon$, with ϵ a small positive quantity.

The second approach changes the set F. If $\mathscr{T}$ is the set of trips, F would change from $F = \{(i,j) \mid i, j \in \mathscr{T}, j.t_0 - i.t_1 - t(i.s_1, j.s_0) \geq 0\}$ to $F = \{(i,j) \mid i, j \in \mathscr{T}, \Delta_{idle} \geq j.t_0 - i.t_1 - t(i.s_1, j.s_0) \geq 0\}$, where $.t_0,\ .t_1$ represent the starting and ending times of a trip, $.s_0,\ .s_1$ are its initial and final stations, t is the travelling time function, and Δ_{idle} is the maximum idling time allowed.

These cost functions prioritize minimizing the total relocating (or rebalancing time) over the maximization of matches. To change this preference to the minimization of the number of vehicles employed, the cost function could be updated by giving the same cost to all feasible edges from F that fulfill the maximum relocating time (or rebalancing time) constriction defined by δ, β. This way, the result of the cost function would be the chose cost, times the number of chainings between trips.

There are several options to implement a dynamic approach that should be tested to see the effect on fleet size, traffic increase and all other parameters considered in previous sections: from making vehicles travel in a random motion through streets, to taking into account the geographical distribution of the demand at each moment of time - as seen in section *Demand analysis* - and sending vehicles to those locations, proportionally to the expected density of requests. The demand distribution should take into account, not only the time of the day, but also the day of the week and the month.

For a dynamic rebalancing approach to be fruitful, it must be applied between iterations of the pooling algorithm. This way, an extra delay may be allowed by changing the rebalancing feasibility condition. If a trip's current and future passengers have not reached their maximum waiting time and delay constrictions, an extra delay on the start of the trip may be applied, so that more connections become feasible for the rebalancing. This delay, however, should not cause the relocating time to violate its maximum.

There are other further steps to be taken to provide an overall more complete approach. For example, the fleet being considered is made up of vehicles of the same capacity; a mixed-size fleet could decrease the total number of vehicles used, and improve the vehicles usage rate. This would need to be added as a parameter for the assignment problem at the rebalancing, to make sure that a vehicle does not attempt to provide service to more passengers than it can actually fit in at once.

In conclusion, applying the pooling and rebalancing codes is a good tool to compute the number of vehicles needed to perform the requests made during the day, for some parameters of the quality of the service to be offered. This would help reduce the fleet size, smooth the traffic flow and decrease the total carbon emissions. However, further improvements would be needed to be able to actually apply it as an implementation on real time.

References

[1] Javier Alonso-Mora et al. "On-demand high-capacity ride-sharing via dynamic trip-vehicle assignment". In: *Proceedings of the National Academy of Sciences* 114.3 (Jan. 2017), pp. 462–467. DOI: 10.1073/pnas.1611675114. URL: https://doi.org/10.1073/pnas.1611675114.

[2] Bruce Schaller. *Empty Seats, Full Streets. Fixing Manhattan's Traffic Problem*. Tech. rep. Schaller Consulting, 2017.

[3] Bruce Schaller. *Unsustainable? The growth of app-based ride sercives and traffic, travel and the future of New York City*. Tech. rep. Schaller Consulting, 2017.

[4] Kalon Kelly. "Casual Carpooling-Enhanced". In: *Journal of Public Transportation* 10.4 (Dec. 2007), pp. 119–130. DOI: 10.5038/2375-0901.10.4.6. URL: https://doi.org/10.5038/2375-0901.10.4.6.

[5] Nelson D. Chan and Susan A. Shaheen. "Ridesharing in North America: Past, Present, and Future". In: *Transport Reviews* 32.1 (Jan. 2012), pp. 93–112. DOI: 10.1080/01441647.2011.621557. URL: https://doi.org/10.1080/01441647.2011.621557.

[6] Sarah Perez. "Uber debuts a "smarter" UberPool in Manhattan". In: *Tech Crunch* (2017). URL: https://techcrunch.com/2017/05/22/uber-debuts-a-smarter-uberpool-in-manhattan/.

[7] Kevin Spieser et al. "Toward a Systematic Approach to the Design and Evaluation of Automated Mobility-on-Demand Systems: A Case Study in Singapore". In: *Road Vehicle Automation*. Springer International Publishing, 2014, pp. 229–245. DOI: 10.1007/978-3-319-05990-7_20. URL: https://doi.org/10.1007/978-3-319-05990-7_20.

[8] R. Tachet et al. "Scaling Law of Urban Ride Sharing". In: *Scientific Reports* 7.1 (Mar. 2017). DOI: 10.1038/srep42868. URL: https://doi.org/10.1038/srep42868.

[9] Niels A.H. Agatz et al. "Dynamic ride-sharing: A simulation study in metro Atlanta". In: *Transportation Research Part B: Methodological* 45.9 (Nov. 2011), pp. 1450–1464. DOI: 10.1016/j.trb.2011.05.017. URL: https://doi.org/10.1016/j.trb.2011.05.017.

[10] Mark Burris and Justin Winn. "Slugging in Houston — Casual Carpool Passenger Characteristics". In: *Journal of Public Transportation* 9.5 (Dec. 2006), pp. 23–40. DOI: 10.5038/2375-0901.9.5.2. URL: https://doi.org/10.5038/2375-0901.9.5.2.

[11] Niels Agatz et al. "Optimization for dynamic ride-sharing: A review". In: *European Journal of Operational Research* 223.2 (Dec. 2012), pp. 295–303. DOI: 10.1016/j.ejor.2012.05.028. URL: https://doi.org/10.1016/j.ejor.2012.05.028.

[12] Frank Spielberg and Phillip Shapiro. "Mating Habits of Slugs: Dynamic Carpool Formation in the I-95/I-395 Corridor of Northern Virginia". In: *Transportation Research Record* 1711 (Jan. 2000), pp. 31–38. DOI: 10.3141/1711-05.

[13] Steve J Beroldo. "Ridematching system effectiveness: a coast-to-coast perspective". In: 1991.

[14] Catherine Morency. "The ambivalence of ridesharing". In: *Transportation* 34.2 (Oct. 2006), pp. 239–253. DOI: 10.1007/s11116-006-9101-9. URL: https://doi.org/10.1007/s11116-006-9101-9.

[15] LeBlanc. *Slugging: The Commuting Alternative for Washington, DC*. 1999.

[16] Philip A. Gruebele. "Interactive System for Real Time Dynamic Multi-hop Carpooling". In: 2008.

[17] Masabumi Furuhata et al. "Ridesharing: The state-of-the-art and future directions". In: *Transportation Research Part B: Methodological* 57 (Nov. 2013), pp. 28–46. DOI: 10.1016/j.trb.2013.08.012. URL: https://doi.org/10.1016/j.trb.2013.08.012.

[18] *Mosek*. URL: https://www.mosek.com/.

[19] Aleksejs Voroncovs; Technische Universität München Mark J. Becker Wolfgang F. Riedl. *The Hungarian Method*. URL: https://www-m9.ma.tum.de/graph-algorithms/matchings-hungarian-method.

[20] James Payor. *Weighted bipartite perfect matching*. URL: https://github.com/jamespayor/weighted-bipartite-perfect-matching.

[21] Jack Edmonds and Richard M. Karp. "Theoretical Improvements in Algorithmic Efficiency for Network Flow Problems". In: *Journal of the ACM (JACM)* 19.2 (Apr. 1972), pp. 248–264. DOI: 10.1145/321694.321699. URL: https://doi.org/10.1145/321694.321699.

[22] Martin Korytak David Fiedler. *Roadmap processing*. URL: https://github.com/aicenter/roadmap-processing.

[23] J.G. Leatham I. Todhunter. *Spherical Trigonometry*. Page 68. Macmillan & Co.Lid., 1914.

[24] NYC Open Data. URL: https://opendata.cityofnewyork.us/.

[25] NYC Open Data. *NYC Taxi Zones*. URL: https://data.cityofnewyork.us/Transportation/NYC-Taxi-Zones/d3c5-ddgc.